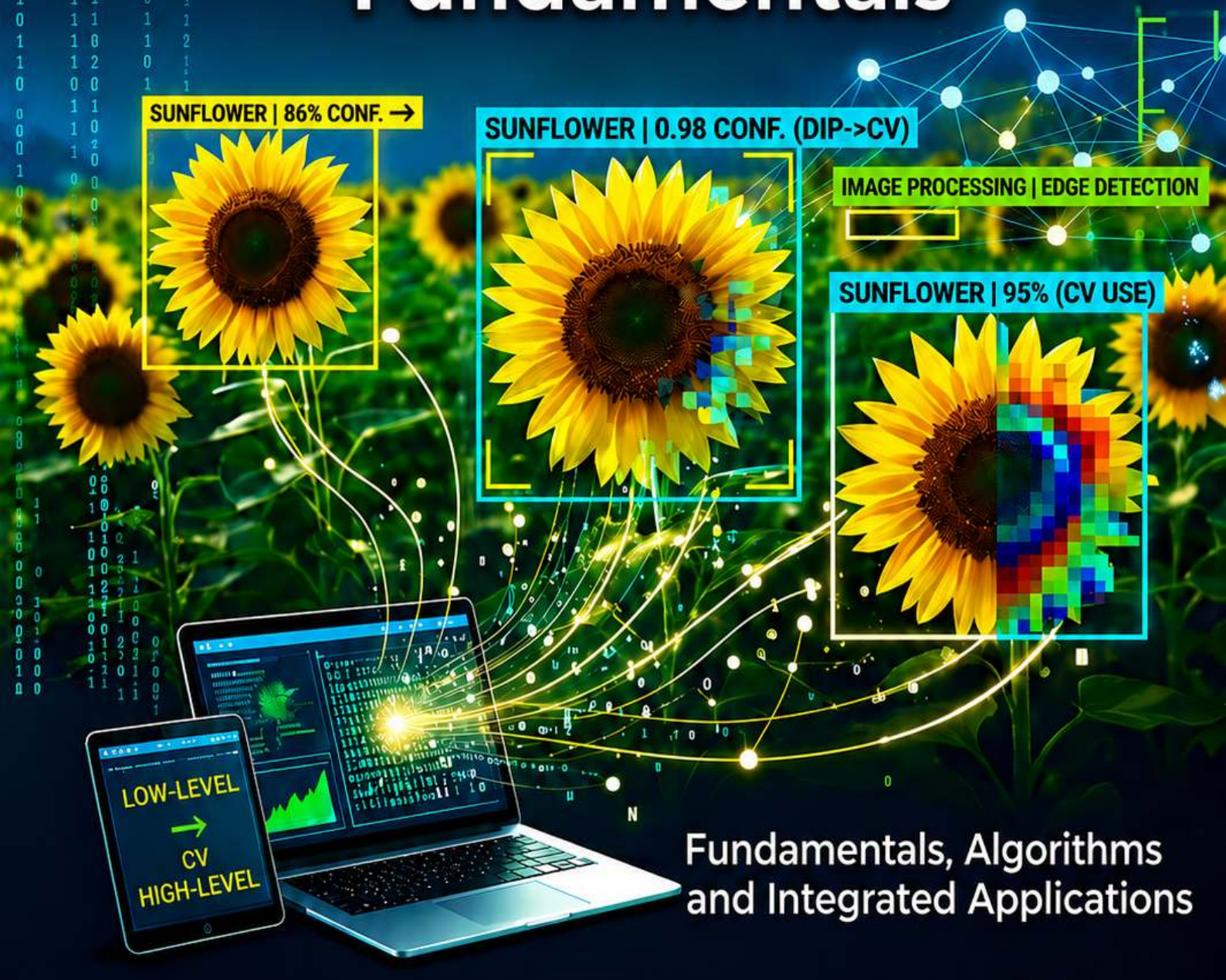
DIP &CV

Digital Image Processing and Computer Vision Fundamentals



Fundamentals, Algorithms and Integrated Applications

AUTHOR: FRANCISCO DE ASSIS ZAMPIROLLI

Digital Image Processing and Computer Vision

Interactive book with C++

Francisco de Assis Zampirolli

Universidade Federal do ABC

September 20, 2026

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

CATALOGING-IN-PUBLICATION DATA
UFABC LIBRARY SYSTEM

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

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

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

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

CDD [XX] ed. – [CDD CODE]

Contents

List of Figures

List of Tables

DIP and CV — C++

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

Organization

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

---

Preface

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

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

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

Context of the first edition

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

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

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

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

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

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

How this book is produced

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Use of Artificial Intelligence tools

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

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

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

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

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

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

The `morph.hpp` library

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

It is the C++ counterpart of `morph.py`: header-only (just `#include "morph.hpp"`), with the **same function names** in the `mm::` namespace — anyone already familiar with `mm.dil()`, `mm.dil0()`, or `mm.gradm()` in Python immediately recognizes `mm::dil()`, `mm::dil0()`, and `mm::gradm()` in C++. A name mismatch between the two versions is treated as a defect of the library, not a design choice.

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

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

- `mm::dil0`: didactic implementation of dilation for **planar structuring elements**, following directly the classic definition of Mathematical Morphology;
- `mm::dil1`: didactic implementation of dilation for **structuring functions (non-planar *kernels*)**, rigorously following its mathematical formulation;
- `mm::dil`: by default, delegates to `mm::dil0()` — the planar didactic version, numerically equivalent to `cv::dilate` for that kind of structuring element. The optimized implementation, based on **OpenCV** (`cv::dilate`), only kicks in if the program is compiled with the `-DMM_USE_OPENCV` flag.

> **!** A deliberate difference from `morph.py`
>
> In Python, `mm.dil()` (no suffix) is **already** the optimized implementation by default. In C++, `mm::dil()` (same name, same signature) only becomes the optimized version if OpenCV is explicitly linked at compile time; without that — which is precisely the default case, including on Moodle/VPL — `mm::dil()` behaves like `mm::dil0()`. This choice keeps the C++ track from depending on OpenCV just to compile and run a simple exercise: `g++ file.cpp -o file` is already enough. Those who want the accelerated version locally can compile with `g++ -DMM_USE_OPENCV file.cpp -o file $(pkg-config --cflags --libs opencv4)`.

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

```cpp
#include "morph.hpp"

int main() {
    mm::Image img = mm::gray(mm::read("lena.jpg"));  // read and convert to grayscale
    mm::Image grad = mm::gradm(img);                 // morphological gradient
    mm::show(grad, "output.png");                    // display (writes to a file)
}
```

Unlike the Python version, `mm::show()` requires an explicit output path: each C++ code cell in the book runs as its own isolated process (compiled and executed via `%%writefile` + `!g++ ...` on Colab), without

the global figure counter that only makes sense within a single Jupyter process.

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

> Explain how the `mm::dil0` function from `morph.hpp` was implemented, showing the code and commenting on each step in a didactic way. Also explain the difference between `mm::dil0()` and `mm::dil()` without the `-DMM_USE_OPENCV` flag.

! Didactic implementations and optimized implementations

In several chapters, the same algorithm is presented in two versions. Functions with a suffix `0`, `1`, etc. (for example, `mm::dil0()` and `mm::dil1()`) are didactic implementations, developed to facilitate understanding of the algorithms and available regardless of the compilation flag. Functions without a suffix (such as `mm::dil()`) use the OpenCV-based optimized implementation **only** when compiled with `-DMM_USE_OPENCV`; otherwise, they behave like the corresponding didactic version.
In activities graded by **Moodle's VPL**, compilation follows the default without `-DMM_USE_OPENCV` — not because of memory limitations (as with scikit-learn/scikit-image in the Python track), but so that no C++ EP depends on OpenCV being installed in the execution environment. In practice, this means that, on VPL, `mm::dil()` and `mm::dil0()` produce exactly the same result. Locally — for example, in **VS Code** or **Google Colab** — students can choose to compile with `-DMM_USE_OPENCV` to compare against the optimized version.

The library's source code is available at:

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

Programming Exercises (EPs) and Automatic Grading

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

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

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

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

Open source

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

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

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

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

Before you begin: Python *Notebooks*

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

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

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

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

Instructor's Guide: Publishing EPs in Moodle

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

Integration with Moodle (VPL)

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

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

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

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

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

Mandatory review before publishing to Moodle

Although automated, the conversion requires instructor review due to limitations of Moodle's TinyMCE/HTML Purifier editor:

- **Mathematical formulas** — TinyMCE discards backslashes (\), corrupting LaTeX equations. For this reason, the Moodle version converts simple formulas to plain HTML (entities such as `&ge;`, `&times;`). Complex formulas (`\begin{cases}`, matrices, integrals) may come out incomplete — review them and, if necessary, rewrite them as text or point to the full version via *link*.
- **External figures** — Supplementary images must be inserted manually into the VPL activity.
- **Simulators** — Work perfectly as long as they use standard JavaScript with *inline* styles and direct DOM manipulation (`getElementById`). **Warning:** if you include LaTeX formulas containing the \ character in Moodle, the simulators may stop working.

How to Publish to Moodle

1. Access the Moodle version of the desired EP:

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

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

Reusing Test Cases

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

Part I

Part I — PDI Fundamentals

Chapter 1

Fundamentals and First Steps

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

Use the C++ language and the `morph.hpp` library, a didactic port of `morph.py` (ZAMPIROLLI, 2025).

1.1 Objectives

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

- Understand the physical and mathematical nature of the digital image $f(x, y)$.
- Identify the bands of the electromagnetic spectrum relevant to DIP.
- Perform basic operations: reading, displaying, and saving images.
- Access and modify pixel intensities individually.
- Apply manual thresholding.
- Set up the C++ development environment.
- Manipulate matrix structures (`std::vector`) without falling into copy/reference pitfalls.

1.2 Before you begin: Interactive *Notebooks*

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

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

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

1.3 Fundamentals

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

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

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

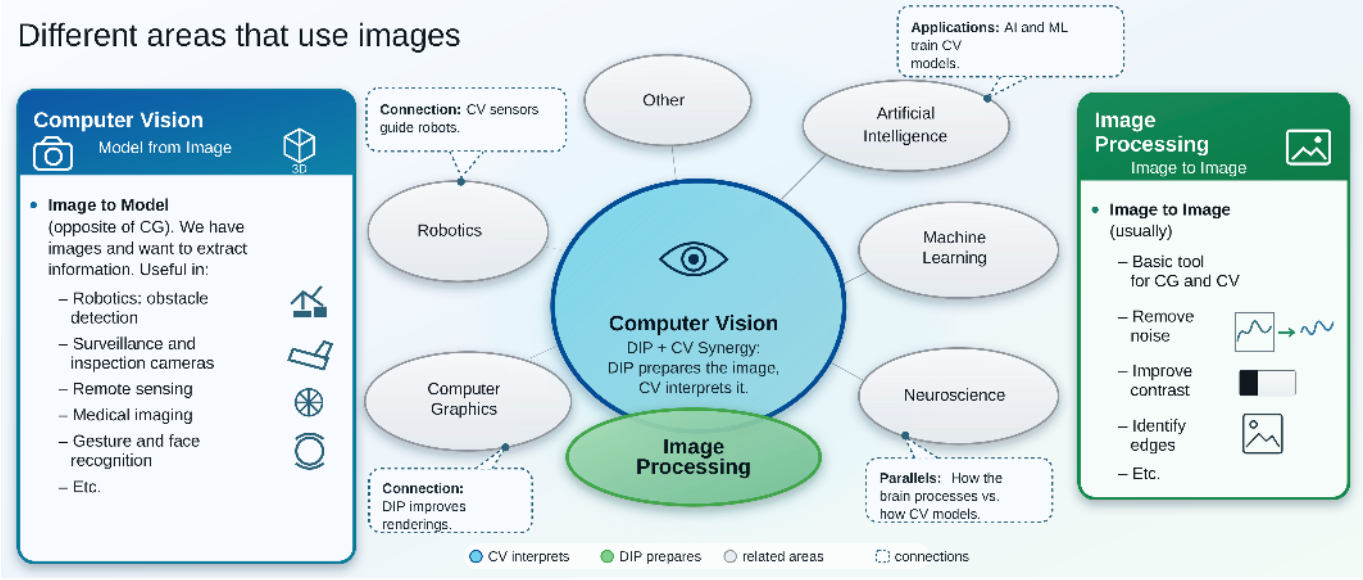


Figure 1.1: Relational diagram detailing the fundamental distinctions, synergies, and interconnections between DIP and CV in the context of image-based systems.

1.3.1 👁️ Computer Vision

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

1.3.2 🖼️ Digital Image Processing (DIP)

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

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

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

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

1.4 PDI Stages

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

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

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

1.5 Image Formation and the Spectrum

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

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

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

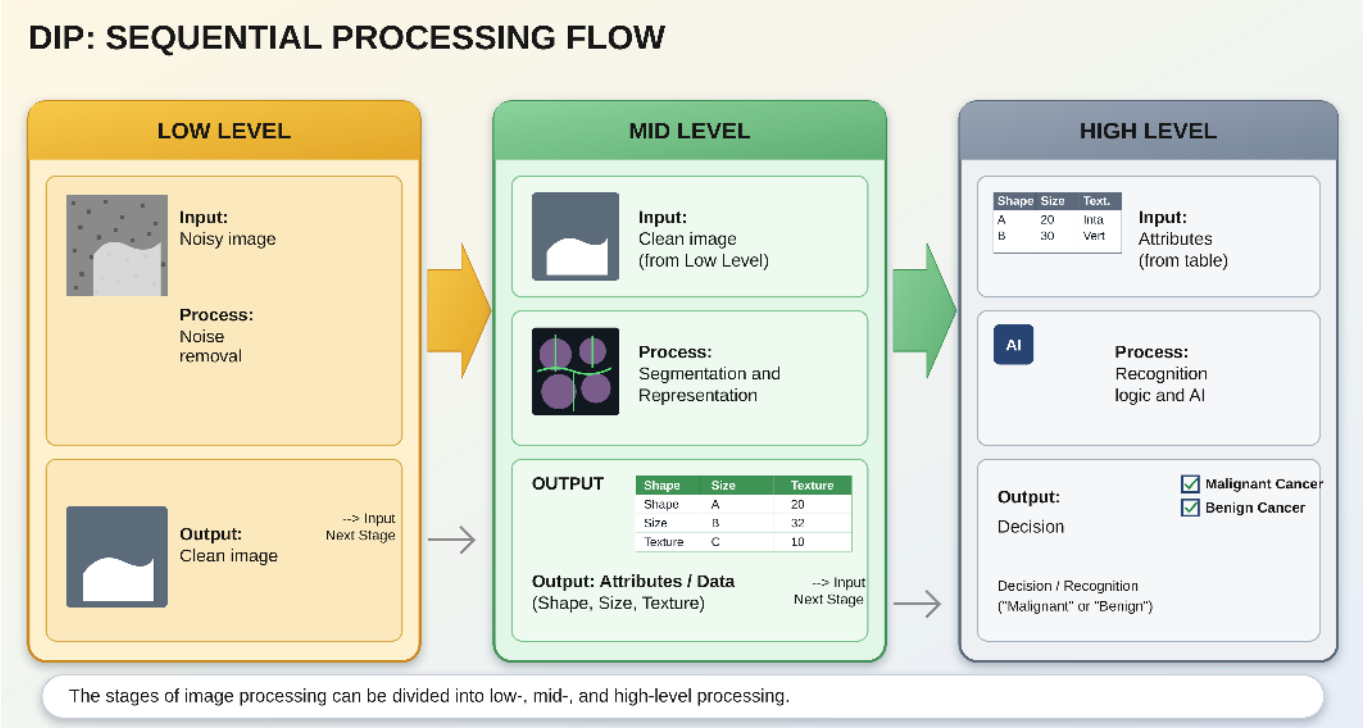


Figure 1.2: Representation of the sequential processing flow: the output of each level becomes the input of the subsequent level.

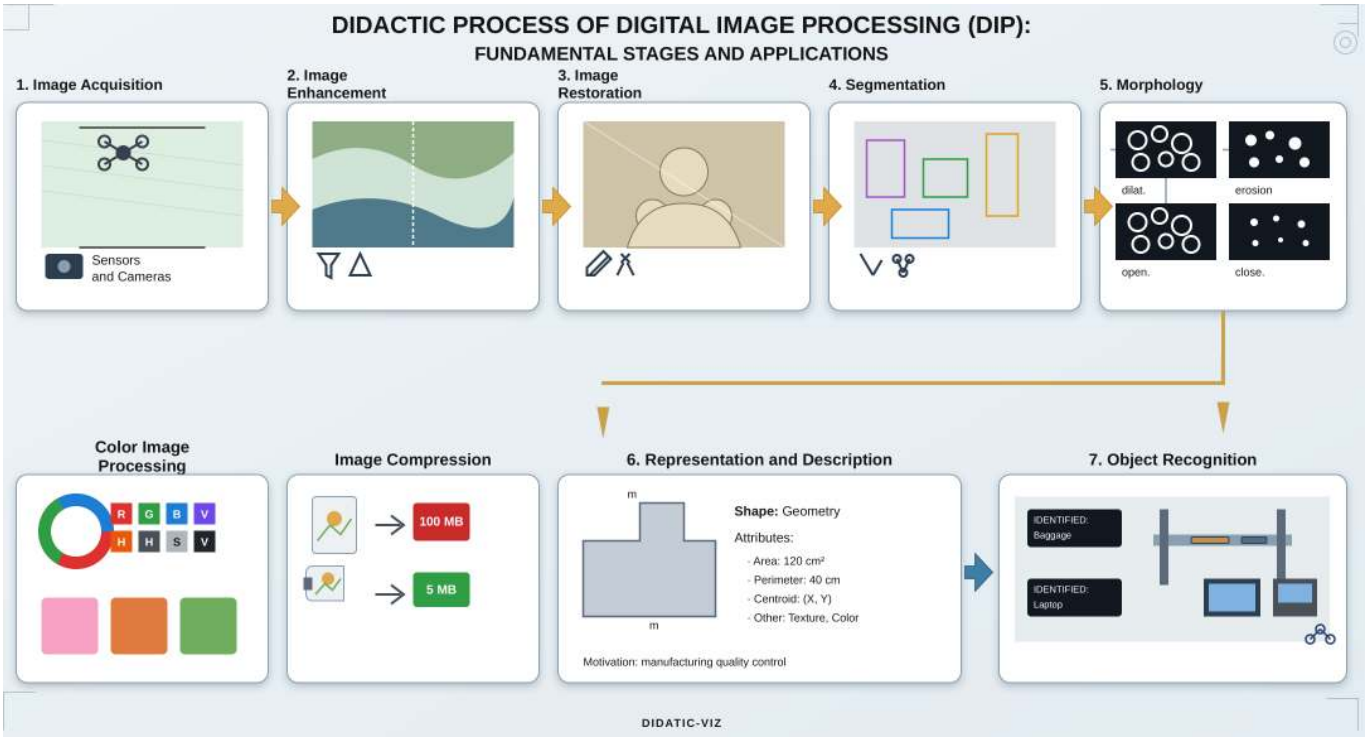


Figure 1.3: Detailed DIP flow: from sensory acquisition to attribute extraction and automated recognition, including color processing and compression.

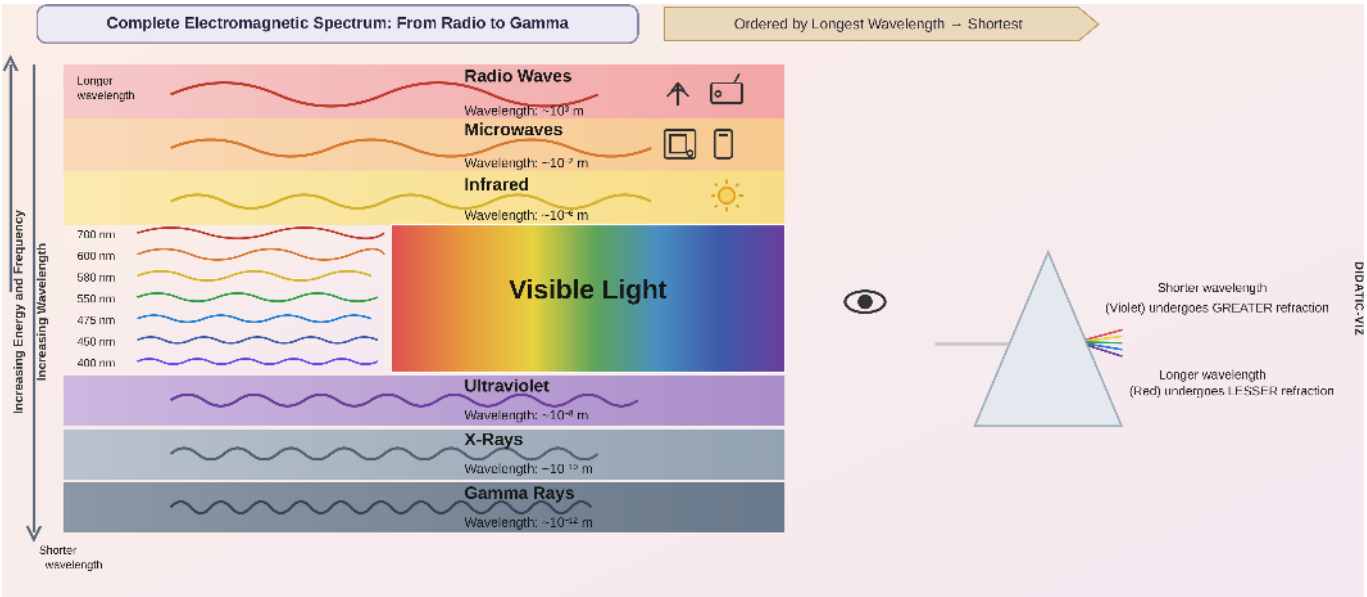


Figure 1.4: Representation of the visible spectrum and its position relative to other electromagnetic radiations, highlighting the variation in wavelengths from 380 nm to 750 nm.

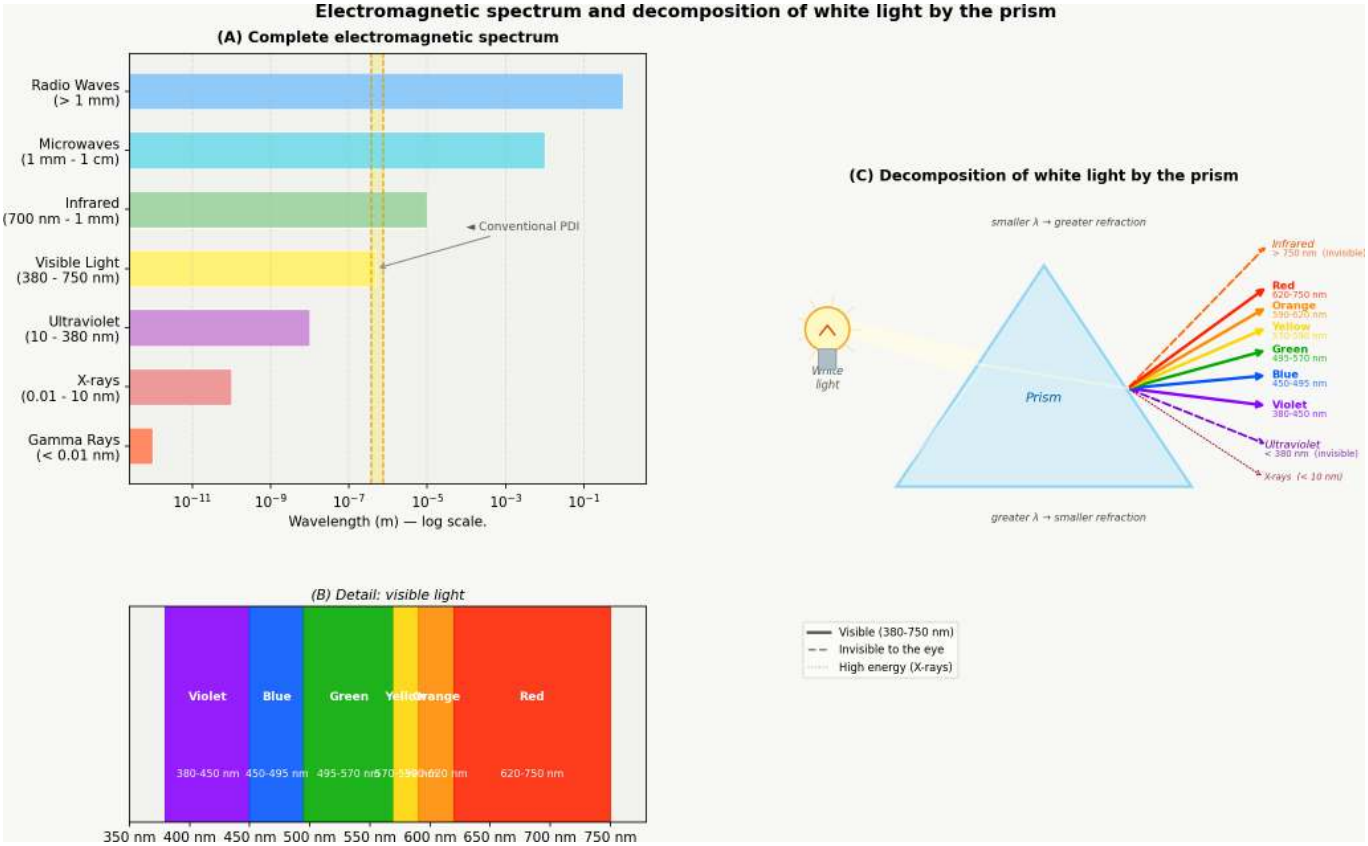


Figure 1.5: (A) Complete electromagnetic spectrum on a logarithmic scale, highlighting the visible range. (B) Detail of visible light (380–750 nm) and its colors. (C) Decomposition of white light by the prism: shorter wavelengths undergo greater refraction, separating UV, visible, and infrared light.

1.6 What is a Digital Image?

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

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

Mathematical Representation

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

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

Where:

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

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

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

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

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

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

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

To enable practical PDI and VC experiments, this book uses C++17 and a minimal set of libraries focused on matrix manipulation, image processing, and result visualization, presented below.

1.7 Environment Setup

This material uses **C++17** and the single-header library **morph.hpp**, originally proposed for Python in Zampirolli (2025). Here, a minimal and didactic version of `morph.py` is used, adapted for straightforward compilation with `g++`, as presented in Table 1.3.

Each code cell is compiled and run as an isolated process (`%%writefile file.cpp` followed by `g++`). Unlike the Python *kernel*, there is no persistent state between cells: the images produced by a cell are saved to files to be read by the next one.

The **Programming Exercises (PEs)** presented at the end of the chapters can be validated by the same `testsuite.py` module used in the Python track, which compiles the solution with `g++` and compares its output with the test cases. Thus, the same test cases (`.cases`) can validate solutions in C++, Python, and other languages, both in the *notebooks* and in **Moodle/VPL**.

Table 1.3: Main libraries and tools used in the C++ track of this book.

Library / Tool	Main function
`g++` (C++17)	Compilation of each code cell
`morph.hpp`	Didactic abstraction of DIP operations
`stb_image.h` / `stb_image_write.h`	Reading/writing PNG/JPEG (without OpenCV)
`testsuite.py`	Execution and automatic validation of PEs

1.8 About `morph.hpp`

`morph.hpp` is a minimal, didactic C++ version of `morph.py`: it implements the functions used in this chapter (`read`, `gray`, `randomImage`, `show`, `write`, `threshold`, `drawImg`), in addition to morphology operations (`dil`/`ero` and variants `dil0`/`ero0`/`dil1`/`ero1`) prepared for the following chapters. There is no guaranteed numerical parity with the Python implementation—the criterion is "it compiles and produces a plausible image," not "a bit-for-bit identical result to Python."

The `stb_image.h`/`stb_image_write.h` libraries (public domain/MIT) are **vendored** alongside the repository—that is, their source code is already copied into the project itself, rather than installed separately via `apt install`—which avoids depending on system packages during compilation on Colab. The `dil()`/`ero()` operations use `cv::dilate`/`cv::erode` when compiled with `-DMM_USE_OPENCV`; without this *flag* (the default, including in Moodle/VPL), they fall back to the equivalent didactic version (`dil1`/`ero1`) without requiring OpenCV.

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # C++ track build artifacts (.cpp, binary, PNGs)

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")

# The kernel is Python even in the C++ track: `mm` (morph.py) is used by the
# simulators, by the display of the figures the C++ binary generates and by the
# mm::Image state between cells. cpp=True also downloads the compiled track
# (morph.hpp + stb_image*.h), used in the #include of the %%writefile *.cpp cells.
import config
config.setup(testsuite=True, cpp=True)
from morph import mm
from testsuite import TestSuite
```

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

1.9 Matrix Fundamentals — Beware of Copying References

Since a digital image can be represented by a matrix, it is important to understand how to correctly create and manipulate matrices. In C++, `std::vector` has **value semantics**: copying the vector copies its

data. Therefore, `std::vector<std::vector<int>> m(3, std::vector<int>(2, 0))` indeed creates three **independent** rows — the fill constructor copies the template row three times.

> ⚠ Beware of Copying References
>
> The pitfall appears when **pointers** or **references** are used instead of copies. In `std::vector<int> linha(2, 0); std::vector<std::vector<int>*> m(3, &linha);`, the three elements of `m` point to the **same** row, and changing `(*m[0])[0]` changes all of them. The same occurs with `auto& linha = m[0];` (reference — modifies the matrix), in contrast to `auto linha = m[0];` (copy — leaves the matrix intact).

To visualize this behavior, one can run the code in Python Tutor (which also executes C++ step by step) and compare the effect of copies and references.

In practice, for digital image processing (DIP), the `mm::Image` structure from `morph.hpp` is used, which also has value semantics: `mm::Image b = a;` copies the pixels, while `mm::Image& b = a;` creates only an alias for the same image. The following code presents different ways of creating synthetic images, whose results are displayed in Figure 1.6..

```
%%writefile tmp/fig_imagens_sinteticas.cpp
#define MM_OUT "tmp/fig_imagens_sinteticas.png"
#include "morph.hpp"
#include <iostream>
#include <string>
#include <vector>
#include <filesystem>

int main() {
    // Criando uma imagem preta (zeros) de 4, 6 pixels
    mm::Image img_preta(4, 6);
    img_preta.at(0, 0) = 255;  // pixel branco no canto superior esquerdo

    // Criando uma imagem branca (255) de 4, 6 pixels
    mm::Image img_branca(4, 6);
    std::fill(img_branca.data.begin(), img_branca.data.end(), 255);
    img_branca.at(3, 5) = 0;  // pixel preto no canto inferior direito

    // Criando uma imagem aleatória para testes (ruído)
    mm::Image img_random = mm::randomImage(4, 6, 255);

    std::cout << "Matriz aleatória gerada:" << std::endl;
    std::cout << mm::drawImg(img_random);

    mm::show(
        {img_preta, img_branca, img_random},
        MM_OUT,
        {
            "Predominantemente preta\n(com 1 pixel branco em (0,0))",
            "Predominantemente branca\n(com 1 pixel preto em (3,5))",
            "Imagem aleatória\n(simulação de ruído)"
        },
        3
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_preta, "tmp/fig_imagens_sinteticas_0.png");
mm::write(img_branca, "tmp/fig_imagens_sinteticas_1.png");
```

```
mm::write(img_random, "tmp/fig_imagens_sinteticas_2.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_imagens_sinteticas.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_imagens_sinteticas.cpp -o tmp/fig_imagens_sinteticas \
  && ./tmp/fig_imagens_sinteticas \
  && test -f "tmp/fig_imagens_sinteticas.png" \
  || echo "  mm::show não gravou tmp/fig_imagens_sinteticas.png"
```

```
Matriz aleatória gerada:
 33 203  36 166 106  36
 64 176  23 221  94 214
  8 216  36 213 153 102
 74 138   2 130  92 102
[1] Predominantemente preta
(com 1 pixel branco em (0,0))
[2] Predominantemente branca
(com 1 pixel preto em (3,5))
[3] Imagem aleatória
(simulação de ruído)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_imagens_sinteticas_0.png"),
            mm.read("tmp/fig_imagens_sinteticas_1.png"),
            mm.read("tmp/fig_imagens_sinteticas_2.png"),
        ],
        titles=[
            'Predominantemente preta\n(com 1 pixel branco em (0,0))',
            'Predominantemente branca\n(com 1 pixel preto em (3,5))',
            'Imagem aleatória\n(simulação de ruído)',
        ],
        cols=3,
        axis=True,
        figsize=(9, 3),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_imagens_sinteticas_0.png (ver a versao Python)")
```

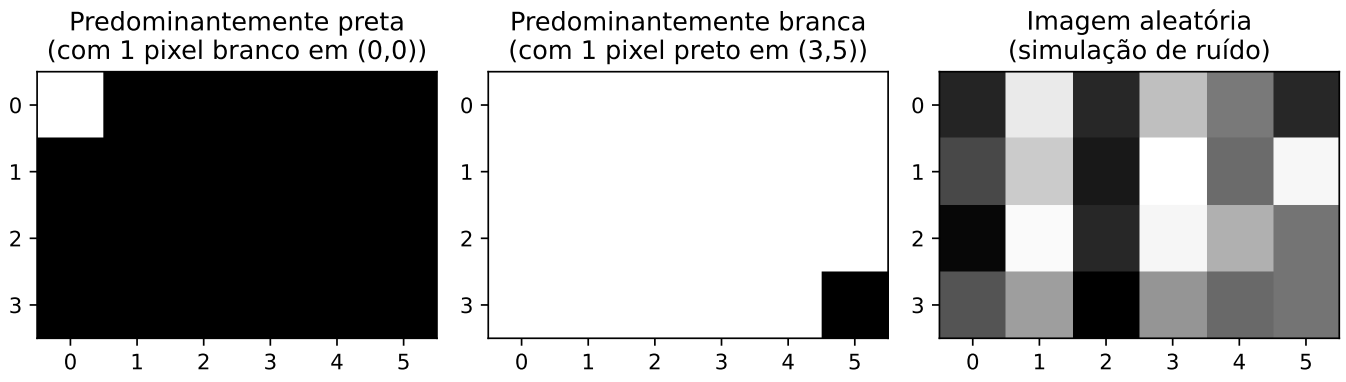


Figure 1.6: Exemplos de imagens sintéticas representadas matricialmente.

1.10 Reading and Displaying Images

In libraries such as OpenCV (`cv::Mat`), digital images are computationally represented as multidimensional arrays. In `morph.hpp`, the `mm::Image` structure adopts a simpler approach: a linear buffer of bytes (`std::vector<unsigned char>`), with explicit height, width, and number of channels.

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

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

```
%%writefile tmp/fig_01_natureza.cpp
#define MM_OUT "tmp/fig_01_natureza.png"
// Compile with: g++ -std=c++17 -I. -o program program.cpp -lcurl

#include "morph.hpp"
#include <iostream>
#include <string>
#include <sys/stat.h>
#include <sys/types.h>
#include <filesystem>

int main() {
    // #| label: fig-01-natureza
    // #| fig-cap: "Mandrill (Mandrillus sphinx) in natural environment. Credit: Julien
    Renoult (CC BY 4.0)."
    // #| echo: true

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

    mm::Image img_obj;

    struct stat st;
    if (stat(caminho.c_str(), &st) != 0) {
        mkdir("imagens", 0755);
        img_obj = mm::read(url);
        mm::write(img_obj, caminho);
    } else {
        img_obj = mm::read(caminho);
    }

    mm::Image img = img_obj;

    std::cout << "Dimensions (H, W, Channels): " << img.h << ", " << img.w << ", " <<
img.channels << std::endl;
    std::cout << "Data type: uint8" << std::endl;

    mm::show(img, MM_OUT, "Exemplo: Captura RGB");


// [pdi:state-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp/state");
mm::write(img, "tmp/state/img_34.png");
// [pdi:state-io:end]
return 0;
}
```

```
Overwriting tmp/fig_01_natureza.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_01_natureza.cpp -o tmp/fig_01_natureza \
  && ./tmp/fig_01_natureza \
  && test -f "tmp/fig_01_natureza.png" \
  || echo "  mm::show não gravou tmp/fig_01_natureza.png"
```

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

```
try:
    mm.show(mm.read("tmp/fig_01_natureza.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_01_natureza.png
    (ver a versao Python)")
```

Figure 1.7: Mandrill (Mandrillus sphinx) em ambiente natural. Crédito: Julien Renoult (CC BY 4.0).

Alternative: Downloading the Image to Local Storage

In environments where direct reading from URLs is unavailable — due to network restrictions, firewall policies, or lack of connectivity — an alternative is to download the image beforehand to the local file system and then load it with `mm::read()`. In the C++ track, `mm::read` also accepts URLs directly (internal download via `curl`/`wget`); this alternative covers the case of downloading once and reusing the local file. In the following example, the file is saved as `mandrill.png`.

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

```
%%writefile tmp/ler_local.cpp
#define MM_OUT "tmp/mandrill_local.png"
#include "morph.hpp"

int main() {
    mm::Image img = mm::read("mandrill.png");   // reads from local file
    mm::show(img, MM_OUT);                      // Example: RGB capture (local file)
```

```
    return 0;
}
```

```
!g++ -I. tmp/ler_local.cpp -o tmp/ler_local && ./tmp/ler_local
```

Explanation

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

Note

The `!` prefix is used in notebook-based environments, such as Jupyter Notebook, JupyterLab, and Google Colab, to run operating system commands directly within code cells. In conventional terminals, the command must be used without the `!` prefix.
If the `wget` utility is not installed, one may alternatively use:

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

1.11 Type Conversion and Thresholding

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

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

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

Conversion to Grayscale

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

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

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

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

Thresholding

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

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

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

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

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

Practical example of conversion and thresholding

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

```
%%writefile tmp/fig_01_processamento_basico.cpp
#define MM_OUT "tmp/fig_01_processamento_basico.png"
#include "morph.hpp"
#include <string>
#include <vector>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img = mm::_read_state("tmp/state/img_34.png");
// [pdi:state-io:end]

    // 1. Convert to Grayscale
    mm::Image img_gray = mm::gray(img);

    // 2. Apply threshold (Pixels > 128 become 255, others 0)
    int limiar = 128;
    mm::Image img_binaria = mm::threshold(img_gray, limiar);

    // Use of the new function
    mm::show(
        std::vector<mm::Image>{img, img_gray, img_binaria},
        MM_OUT,
        std::vector<std::string>{"Original", "Grayscale", "Binary (T=" +
std::to_string(limiar) + ")"},
        3
    );


// [pdi:state-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp/state");
mm::write(img_gray, "tmp/state/img_gray_38.png");
// [pdi:state-io:end]

// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img, "tmp/fig_01_processamento_basico_0.png");
mm::write(img_gray, "tmp/fig_01_processamento_basico_1.png");
mm::write(img_binaria, "tmp/fig_01_processamento_basico_2.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_01_processamento_basico.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_01_processamento_basico.cpp -o tmp/fig_01_processamento_basico \
  && ./tmp/fig_01_processamento_basico \
  && test -f "tmp/fig_01_processamento_basico.png" \
  || echo "  mm::show não gravou tmp/fig_01_processamento_basico.png"
```

```
[1] Original
[2] Grayscale
[3] Binary (T=128)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_01_processamento_basico_0.png"),
            mm.read("tmp/fig_01_processamento_basico_1.png"),
            mm.read("tmp/fig_01_processamento_basico_2.png"),
        ],
        titles=[
            'Original',
            'Tons de Cinza',
            'Binária (T=128)',
        ],
        cols=3,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_01_processamento_basico_0.png (ver a versao Python)")
```

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

1.12 Thresholding using the Otsu method

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

```
mm::Image img_bin_fixo = mm::threshold(img_gray, 128);
```

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

The code below compares manual thresholding ($T = 128$) with automatic thresholding (Otsu). Otsu binarization is performed with `mm::threshold(img_gray)`; since the minimal API of `morph.hpp` does not

return the computed T, the value displayed in the figure label is retrieved with `cv2.threshold` at the display stage (same algorithm, same T).

```
%%writefile tmp/fig_01_otsu.cpp
#define MM_OUT "tmp/fig_01_otsu.png"
#include "morph.hpp"
#include <iostream>
#include <string>
#include <vector>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_38.png");
// [pdi:state-io:end]

    // Limiarização com T fixo (manual)
    int T_fixo = 128;
    mm::Image img_bin_fixo = mm::threshold(img_gray, T_fixo);

    // Limiarização pelo método de Otsu (T automático)
    // cv2.threshold(img_gray, 0, 255, THRESH_BINARY + THRESH_OTSU)
    // maps to: T_otsu = mm::otsu(img_gray); img_bin_otsu = mm::threshold(img_gray);
    int T_otsu = mm::otsu(img_gray);
    mm::Image img_bin_otsu = mm::threshold(img_gray);
    std::cout << "Limiar calculado por Otsu: T = " << T_otsu << "\n";
    // ou simplesmente:
    // img_bin_otsu = mm::threshold(img_gray);

    // Exibição lado a lado
    mm::show(
        {img_gray, img_bin_fixo, img_bin_otsu},
        MM_OUT,
        {"Tons de Cinza", "Binária (T=" + std::to_string(T_fixo) + ")",
         "Binária (Otsu, T=" + std::to_string(T_otsu) + ")"},
        3
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray, "tmp/fig_01_otsu_0.png");
mm::write(img_bin_fixo, "tmp/fig_01_otsu_1.png");
mm::write(img_bin_otsu, "tmp/fig_01_otsu_2.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_01_otsu.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_01_otsu.cpp -o tmp/fig_01_otsu \
  && ./tmp/fig_01_otsu \
  && test -f "tmp/fig_01_otsu.png" \
  || echo "  mm::show não gravou tmp/fig_01_otsu.png"
```

```
Limiar calculado por Otsu: T = 95
[1] Tons de Cinza
[2] Binária (T=128)
[3] Binária (Otsu, T=95)
```

```python
try:
    T_otsu = mm.otsu(mm.read("tmp/fig_01_otsu_0.png", grayscale=True))
    mm.show(
        [
            mm.read("tmp/fig_01_otsu_0.png"),
            mm.read("tmp/fig_01_otsu_1.png"),
            mm.read("tmp/fig_01_otsu_2.png"),
        ],
        titles=[
            'Tons de Cinza',
            'Binária (T=128)',
            f"Binária (Otsu, T={T_otsu})",
        ],
        cols=3,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_01_otsu_0.png (ver
    a versao Python)")
```

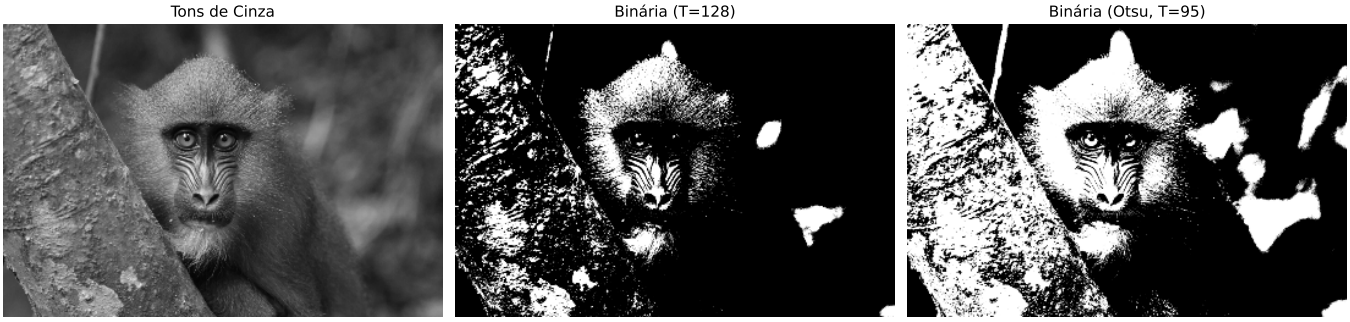


Figure 1.9: Comparação entre limiarização manual (T=128) e automática (Otsu) sobre a imagem em tons de cinza.

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

Simplicity of the `morph.hpp` library

While OpenCV requires the full call:

```cpp
cv::Mat img_bin;
double T_otsu = cv::threshold(img_gray, img_bin, 0, 255,
                              cv::THRESH_BINARY | cv::THRESH_OTSU);
```

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

1.13 Pixel Access

In `morph.hpp`, the image is an `mm::Image` with a linear buffer `data` (row after row, interleaved channels) and the `at(y, x, c)` method for individual access, following the matrix convention of **row** (Y-axis) and **column** (X-axis): `img.at(row, column)` for grayscale and `img.at(row, column, channel)` for each channel of an RGB image.

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

Pixel access in C++ (`mm::Image`):

- **Grayscale:** `img_gray.at(r, c)` returns an `unsigned char` (0 to 255) with the grayscale intensity at the pixel.
- **RGB:** `img.at(r, c, 0)`, `img.at(r, c, 1)`, `img.at(r, c, 2)` access the **R**, **G**, and **B** channels — one channel index at a time; there is no `[R, G, B]` vector.
- 3×3 **neighborhood:** traversed by two nested loops over `img_gray.at(r + dy, c + dx)`, with `dy`, `dx` in $\{-1, 0, 1\}$ — there is no slicing as in NumPy. This scan is the basis for spatial filters and convolutions.
- There is no interactive plotting (equivalent to `plt.plot`): to mark the pixel position, the following code cell **draws a hollow red square** around it on a copy of the image (`marcada = img.copy()`, then `marcada.at(...)`) and displays it with `mm::show`.

Indexing is *zero-based*: `(0,0)` is the top-left corner. The first dimension controls the **height** (rows/Y) and the second the **width** (columns/X).

1.14 Summary

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

- **Digital image** = function $f(x, y)$ that maps coordinates to intensities (scalar or vectorial).
- **Domain**: finite set $\mathbb{E} = \{(x, y) \in \mathbb{Z}^2 \mid 0 \leq x < L,\ 0 \leq y < H\}$.
- **Main types**: binary ($\mathcal{V} = \{0, 255\}$), grayscale ($\mathcal{V} = [0, 255]$), and RGB ($\mathcal{V} = [0, 255]^3$).
- **Thresholding** converts grayscale into binary; **Otsu's method** automatically determines the cutoff value by maximizing between-class variance.
- The `morph.hpp` (or `mm::`) library offers didactic functions for basic DIP operations, such as `mm::gray()`, `mm::threshold()`, and `mm::show()` (overloaded for a single image or multiple images).
- **Copy trap**: `std::vector` has value semantics, but pointers/references shared between the "rows" of a matrix reintroduce the same problem as NumPy — prefer `mm::Image`, which also copies by value.
- Pixel access via `img.at(row, column)`, with ***zero-based*** indexing.

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

1.15 🤖 Using Gemini Notebook as a Complementary Tutor

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

🎓 Study with the Intelligent Tutor

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

```
# Not yet ported to this language in this version - conceptual reference in Python.

# 1. Coordinates of the target pixel (row, column)
r, c = 600, 800

# 2. Direct access to pixel values
pixel_cinza = img_gray[r, c]
print(f"Target pixel ({r}, {c}):")
print(f"  Grayscale (scalar) : {pixel_cinza}")
print(f"  Colored (RGB channels)   : R={img[r, c, 0]}, G={img[r, c, 1]}, B={img[r, c, 2]}")

# 3. 3x3 neighborhood around (r, c); the central pixel goes in brackets
print("3x3 neighborhood matrix (grayscale):")
for dy in range(-1, 2):
    linha = ""
    for dx in range(-1, 2):
        v = img_gray[r + dy, c + dx]
        if dy == 0 and dx == 0:
            linha += f"[{v:3d}]"
        else:
            linha += f" {v:3d} "
    print("  " + linha)
# 4. Marks the pixel position with a hollow red square (border of
#    3 px) on a copy of the image and displays (equivalent to the matplotlib dot).
marcada = img.copy()
lado = 20   # half-side of the square, in pixels
esp = 5     # border thickness

for x in range(c - lado, c + lado + 1):
    for w in range(esp):
        marcada[r - lado + w, x, 0] = 255
        marcada[r - lado + w, x, 1] = 0
        marcada[r - lado + w, x, 2] = 0
        marcada[r + lado - w, x, 0] = 255
        marcada[r + lado - w, x, 1] = 0
        marcada[r + lado - w, x, 2] = 0

for y in range(r - lado, r + lado + 1):
    for w in range(esp):
        marcada[y, c - lado + w, 0] = 255
        marcada[y, c - lado + w, 1] = 0
        marcada[y, c - lado + w, 2] = 0
        marcada[y, c + lado - w, 0] = 255
        marcada[y, c + lado - w, 1] = 0
        marcada[y, c + lado - w, 2] = 0

mm.show(marcada, title=f"Pixel location ({r}, {c})")
```

Figure 1.10

🌐 **Language and Programming Language**

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

⚠️ **Notice about AI-Generated Content**

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

Available Platform Features

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

The main features include:

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

1.16 Exercise List

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

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

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

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

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

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

5. (15%) In the context of the didactic library `mm` discussed in the chapter, answer:
   - a) (7.5%) How do you access the pixel value at position (row=50, column=60) of a grayscale image `img_gray`?
   - b) (7.5%) What is the advantage of using `mm::threshold(img_gray)` without passing the threshold? Compare it with the equivalent call in OpenCV (`cv::threshold`).
6. (15%) What do the fields `h`, `w`, and `channels` of an `mm::Image` represent for an RGB image? Give a concrete example with a 640×480 pixel image.

1.17 Chapter References

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

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

1.18 1.7 Proposed Exercises (EPs)

EP1.1. Consider the digital image $f(x, y)$ of dimensions $M \times N$, each pixel is quantized with k bits. Determine the number of distinct colors and the memory size needed to store this image. Express the memory size in terms of M, N, and k. Then, calculate the memory requirement for an image of size 512×512 with $k = 8$ bits per pixel.

EP1.2. Using the concepts of image sampling and quantization, explain the effect known as "aliasing." Describe a practical situation in digital imaging where this effect may occur and suggest a method to minimize it.

EP1.3. Compare the RGB and HSV color models. For each, discuss the advantages and disadvantages in image processing applications. Illustrate an example where converting from RGB to HSV is beneficial.

EP1.4. Implement a function in Python that performs histogram equalization on a grayscale image (use NumPy and OpenCV). Apply the function to a low-contrast image and display the original and resulting images alongside their histograms.

EP1.5. Consider a spatial filter with a mask of size 3×3 defined by the kernel:

$$h(x, y) = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

(a) Classify this filter as low-pass, high-pass, or band-pass. Justify your answer. (b) Apply this filter to a 5×5 artificial image (define it) and present the result. (c) Comment on the effect of applying the filter repeatedly.

EP1.6. Suppose an image has noise with a Gaussian probability density function (PDF) with zero mean and variance σ^2. Propose an image restoration approach using linear filtering to reduce the noise and discuss its limitations.

EP1.7. Using the Fourier transform, explain the relationship between the spatial domain and the frequency domain for a digital image. Describe how to perform low-pass filtering in the frequency domain and compare it with filtering in the spatial domain.

EP1.8. Develop a Python script that detects edges in a grayscale image using the Sobel operator. Show the steps of computing the gradient magnitude and direction. Apply the script to an example image and present the results.

EP1.9. Consider image compression via the discrete cosine transform (DCT), as used in JPEG. Explain the steps of compression and decompression. Discuss the trade-off between compression rate and image quality.

EP1.10. Discuss the main differences between lossless and lossy image compression. Give an example of an application where each type is most appropriate.

1.19 💻 Practical Part with Programming Exercises

1.19.1 🎯 Objective of this Notebook

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

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

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

1.19.2 § Step-by-Step Instructions

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

1.19.2.1 Environment Preparation

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

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

1.19.2.2 Writing the Code

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

Example: `%%writefile EP01_01.py`

1.19.2.3 *Download*

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

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # C++ track build artifacts (.cpp, binary, PNGs)

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")

# The kernel is Python even in the C++ track: `mm` (morph.py) is used by the
# simulators, by the display of the figures the C++ binary generates and by the
```

```
# mm::Image state between cells. cpp=True also downloads the compiled track
# (morph.hpp + stb_image*.h), used in the #include of the %%writefile *.cpp cells.
import config
config.setup(testsuite=True, cpp=True)
from morph import mm
from testsuite import TestSuite
```

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

1.19.2.4 Running the Tests

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

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

Replace the extension according to the language used:

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

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

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

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

1.19.3 ⚠️ Important: Rules and Best Practices

1.19.3.1 🔹 About Data Input

Your program must read from standard input (keyboard).

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

1.19.3.2 🔹 AI Configuration in Colab

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

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

1.19.3.3 🔹 Academic Integrity (Plagiarism)

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

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

1.19.4 EP01_01 📏 Three distance metrics in DIP

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

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

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

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

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

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

📌 **Important**:

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

1.19.4.1 🖼️ Why does this matter? – Computational cost

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

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

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

1.19.4.2 📋 Task (specification for VPL)

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

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

1.19.4.3 📌 Examples

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

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

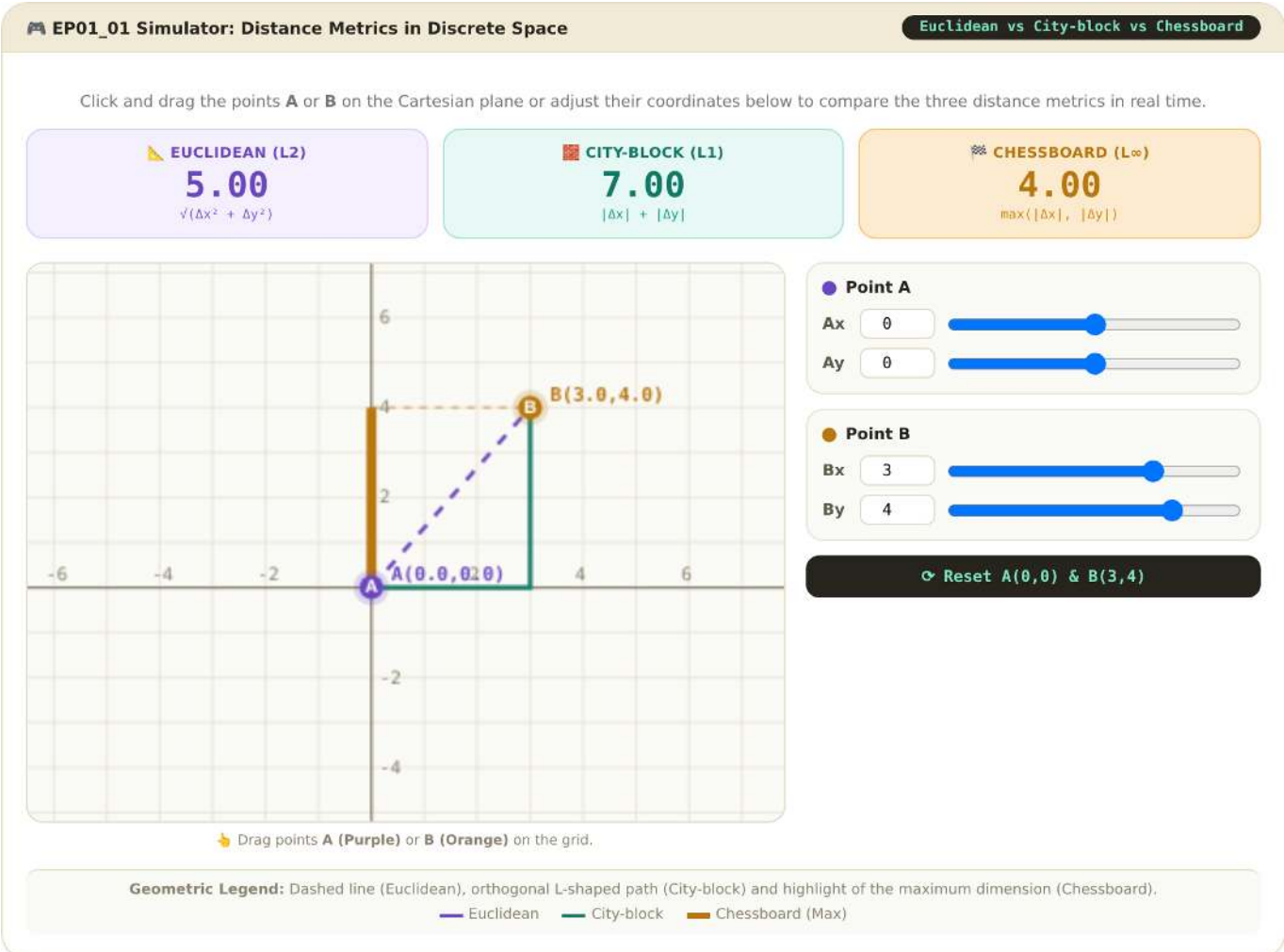
```
%%writefile tmp/mm_out_1.cpp
// Compile: g++ -O2 -std=c++17 benchmark.cpp -o benchmark && ./benchmark
#include <iostream>
#include <iomanip>
#include <cmath>
#include <chrono>

const int N = 50'000'000;

double apenas_soma() {
    double a = 3.0, b = 4.0;
    return a + b;
}

double soma_e_sqrt() {
    double a = 3.0, b = 4.0;
    return std::sqrt(a*a + b*b);
}

int main() {
    auto start_soma = std::chrono::high_resolution_clock::now();
    volatile double sum;
    for (int i = 0; i < N; ++i) {
        sum = apenas_soma();
    }
    auto end_soma = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> t_soma = end_soma - start_soma;

    auto start_sqrt = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < N; ++i) {
        sum = soma_e_sqrt();
    }
    auto end_sqrt = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> t_sqrt = end_sqrt - start_sqrt;

    // Prevent optimization removal
```

```
    if (sum == 12345.6789) std::cout << sum;

    std::cout << std::fixed << std::setprecision(3);
    std::cout << "Soma simples       : " << t_soma.count() << " s\n";
    std::cout << "Soma + sqrt        : " << t_sqrt.count() << " s\n";
    std::cout << "Razão (sqrt/soma)  : " << std::setprecision(2)
              << (t_sqrt.count()/t_soma.count()) << "x\n";

    return 0;
}
```

```
Overwriting tmp/mm_out_1.cpp
```

```
!g++ -I. -std=c++17 tmp/mm_out_1.cpp -o tmp/mm_out_1 \
  && ./tmp/mm_out_1
```

```
Soma simples       : 0.246 s
Soma + sqrt        : 0.456 s
Razão (sqrt/soma)  : 1.85x
```



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

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

1.19.4.4 🐍 Python

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

Example cell:

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

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

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

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

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

```
Overwriting EP01_01.py
```

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

# %run -i EP01_01.py
```

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

```
5.66
8.00
4.00
```

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

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

1.19.4.5 ☕ Java

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

```
%%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>
```

1.19.4.6 💻 C

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

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

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

instalar_gcc()
```

```
  GCC already available.
```

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

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

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

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

```
    return 0;
}
```

```
Overwriting EP01_01.c
```

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

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

```
5.66
8.00
4.00
```

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

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

1.19.4.7 💻 C++

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

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

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

instalar_gpp()
```

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

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

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

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

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

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

    return 0;
}
```

```
Overwriting EP01_01.cpp
```

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

```
5.66
8.00
4.00
```

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

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

1.19.4.8 🌐 JavaScript (Node.js)

For JavaScript, use `%%writefile` to create the file and run it with Node:

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

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

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

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

```
Overwriting EP01_01.js
```

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

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

1.19.4.9 📊 R

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

Example cell:

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

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

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

instalar_r()
```

```
  R is already available.
```

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

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

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

```
Overwriting EP01_01.r
```

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

```
5.66
8.00
4.00
```

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

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

1.19.5 EP01_02 📊 Predictive Performance — ML Metrics in CV

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

1.19.5.1 🧠 Why does the right metric matter?

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

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

1.19.5.2 📊 The Confusion Matrix

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

Task:

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

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

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

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

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

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

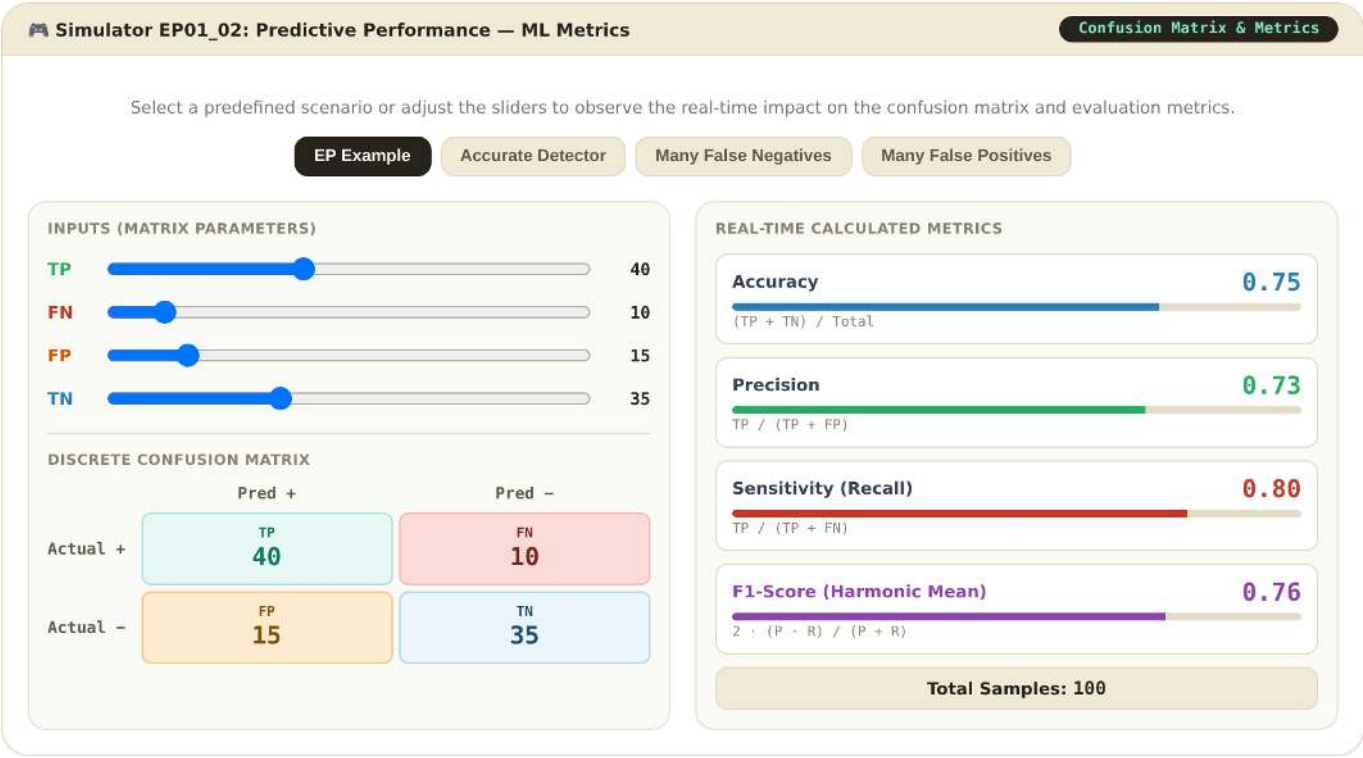
📌 **Important**:

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

1.19.5.3 📌 Example Execution

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

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



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

Figure 1.12: EP01_02 Simulator: Predictive Performance — ML Metrics

```
%%writefile EP01_02.cpp
// your solution
```

```
Overwriting EP01_02.cpp
```

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

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

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

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

1.19.6.1 🧠 Why is mAP the standard metric?

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

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

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

References: Roboflow — mAP · Explanatory video

1.19.6.2 How mAP is calculated — step by step

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

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

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

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

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

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

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

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

1.19.6.3 📋 Task

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

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

- Confusion Matrix (TP, FN, FP, TN)

- Accuracy, Precision, Recall, and F1-Score

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

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

1.19.6.4 📌 Important

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

1.19.6.5 📌 Example Run

Input	Expected Output
7 0 0.94 1 0.80 1 0.69 0 0.67 1 0.30 1 0.15 1 0.15	# METRICS FOR THRESHOLD 0.85 # Confusion Matrix: TP = 0, FN = 5 FP = 1, TN = 1 Evaluation Metrics: Accuracy: 0.14 Precision: 0.00 Recall: 0.00 F1-Score: 0.00 # METRICS FOR ALL THRESHOLDS # Precisions: 0.00, 0.00, 0.00, 0.50, 0.50, 0.50, 0.50, 0.50, 0.60, 0.71, 0.71 Monotonic Precisions: 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71 Recalls: 0.00, 0.00, 0.00, 0.20, 0.40, 0.40, 0.40, 0.40, 0.60, 1.00, 1.00 mAP: 0.71

1.19.6.6 🐍 Tip for calculating AP (with trapezoid rule)

```
def calcular_AP(verdades, confiancas, limiares):
    m = len(limiares)
    precisoes = [0.0] * m
    sensibilidades = [0.0] * m
    for i in range(m):
        p, s = calcular_metricas(verdades, confiancas, limiares[i])
        precisoes[m-1-i] = p
        sensibilidades[m-1-i] = s
    prec_mono = precisoes.copy()
    for i in range(m-2, -1, -1):
        if prec_mono[i] < prec_mono[i+1]:
            prec_mono[i] = prec_mono[i+1]
    AP = 0.0
    for i in range(1, m):
        # Trapezoid rule: average of heights times the base
        area_trapezio = (prec_mono[i-1] + prec_mono[i]) / 2.0
        AP += area_trapezio * (sensibilidades[i] - sensibilidades[i-1])
```

```
    return precisoes, prec_mono, sensibilidades, AP
```

```
%%writefile EP01_03.cpp
// your solution
```

```
Overwriting EP01_03.cpp
```

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

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

1.19.7 EP01_04 🖼️ Reading and Information from a Matrix Image

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

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

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

📌 **Important**:

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

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

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

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

1.19.7.2 📋 Task (VPL specification)

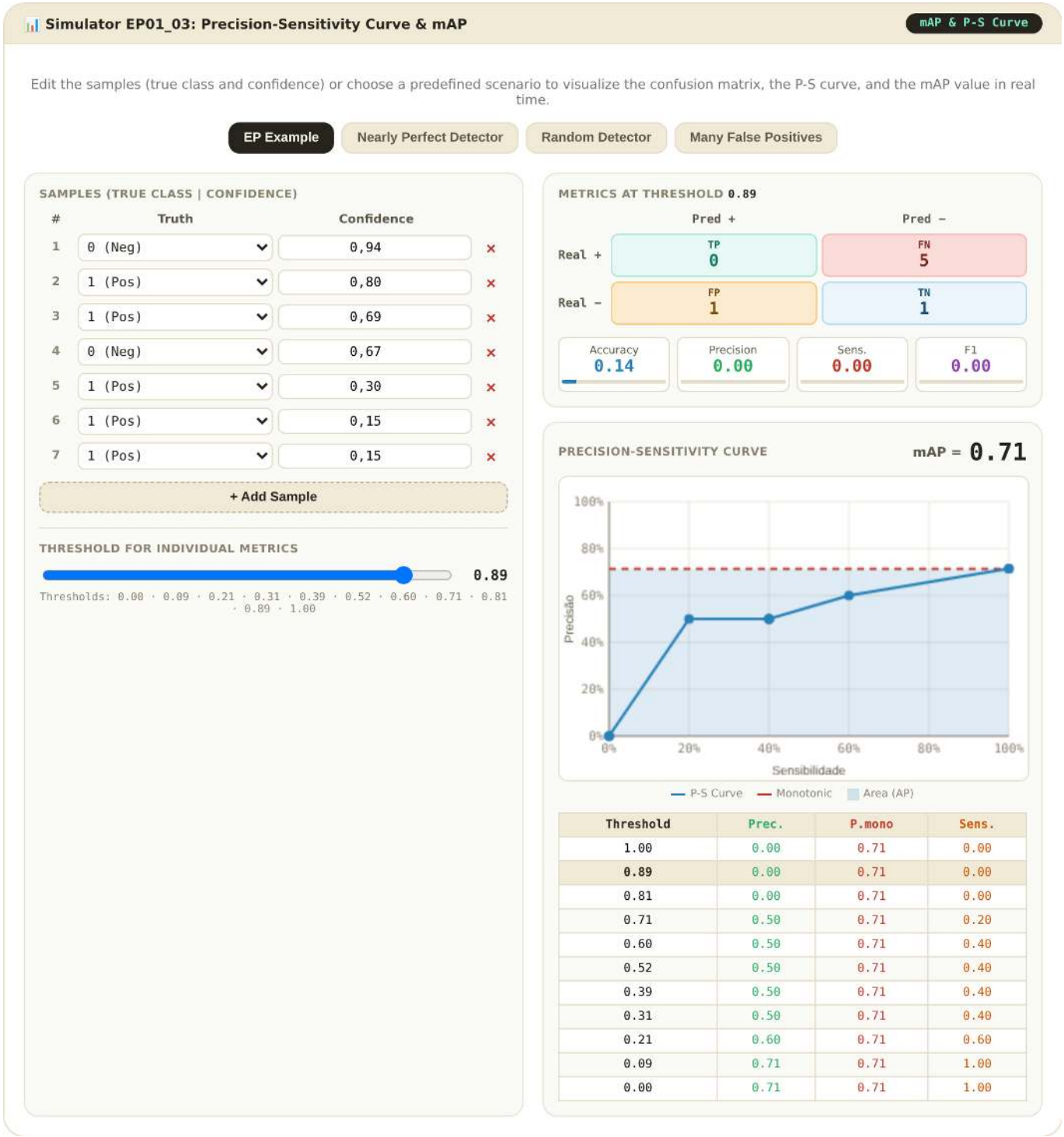
Input:

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

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

The following lines contain the elements of the matrix.

Output:



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

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

Five lines formatted according to the example:

`Linhas: L`

`Colunas: C`

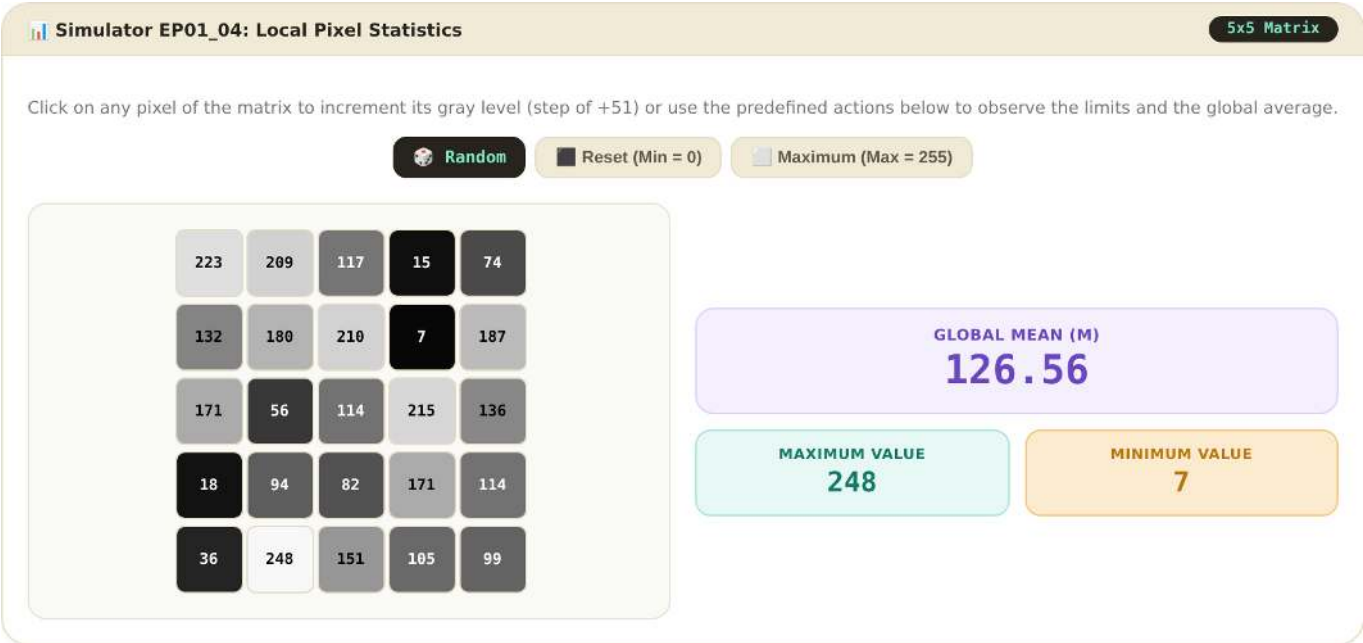
`Max: V`

`Min: V`

`Media: V.VV`

1.19.7.3 📌 Examples

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



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

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

```
%%writefile EP01_04.cpp
// your solution
```

```
Overwriting EP01_04.cpp
```

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

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

1.19.8 EP01_05 🔄 Negative of a Grayscale Image

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

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

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

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

📌 **Important**:

- The values in each row of the output must be separated by a **single space**.
- The output must contain **only the numbers** of the resulting matrix.
- See an interactive simulator for this problem at **Figure 1.15** (real-time comparison between the original matrix and its negative).

1.19.8.1 🧠 Why Does This Matter? – Intensity Inversion

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

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

1.19.8.2 📋 Task (Specification for VPL)

Input:

The first line contains the integer `L`.

The second line contains the integer `C`.

The following lines contain the elements of the matrix.

Output:

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

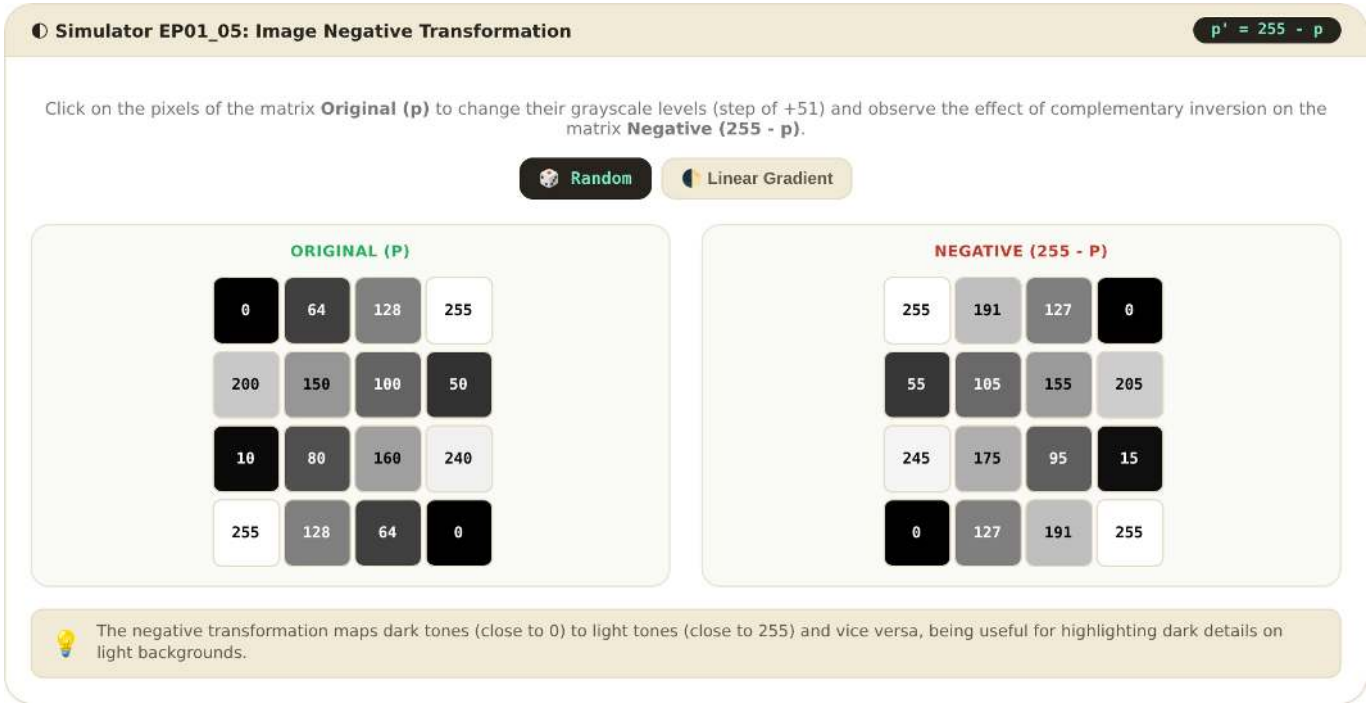
1.19.8.3 📌 Examples

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

```
%%writefile EP01_05.cpp
// your solution
```

```
Overwriting EP01_05.cpp
```

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



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

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

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

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

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

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

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

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

📌 **Important**:

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

1.19.9.1 🧠 Why not just use the average?

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

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

1.19.9.2 📋 Task (VPL specification)

Input:

The first line contains the integer `L`.

The second line contains the integer `C`.

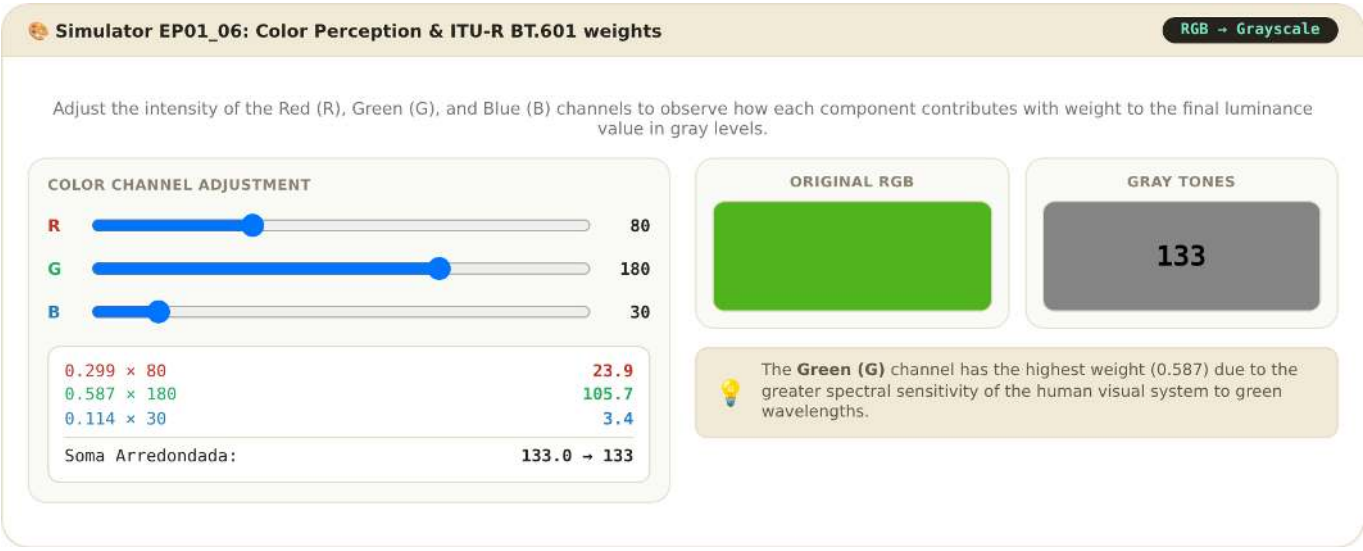
The following lines contain triples of integers `R G B` for each pixel.

Output:

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

1.19.9.3 📌 Examples

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



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

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

```
%%writefile EP01_06.cpp
// your solution
```

```
Overwriting EP01_06.cpp
```

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

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

1.19.10 EP01_07 ⚫ Manual Thresholding: Binary Image

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

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

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

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

📌 **Important**:

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

1.19.10.1 🧠 What is Segmentation?

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

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

1.19.10.2 📋 Task (specification for VPL)

Input:

The first line contains the integer `L`.

The second line contains the integer `C`.

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

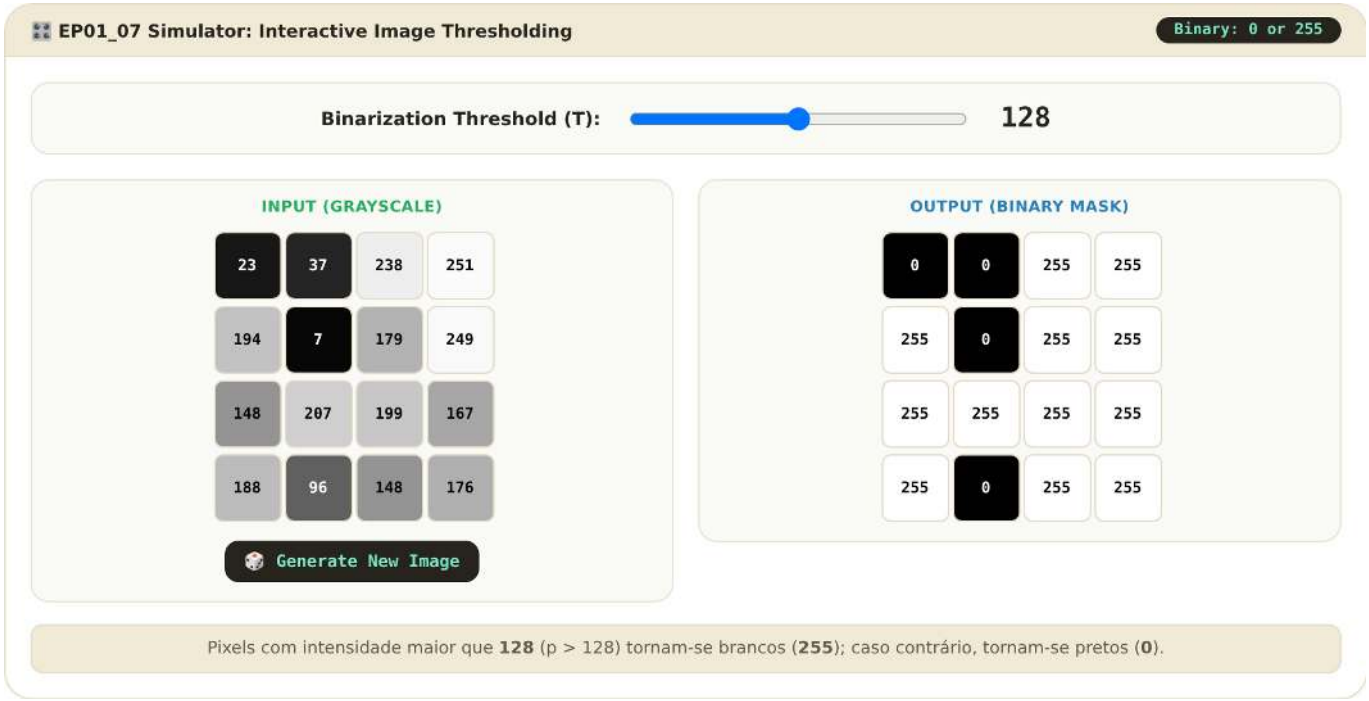
The following lines contain the elements of the matrix.

Output:

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

1.19.10.3 📌 Examples

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



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

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

```
%%writefile EP01_07.cpp
// your solution
```

```
Overwriting EP01_07.cpp
```

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

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

1.19.11 EP01_08 🎨 Intensity Range Remapping

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

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

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

- Print the resulting matrix with the new intensity values.

📌 **Important**:

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

1.19.11.1 🧠 Conditional Pixel Transformation

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

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

1.19.11.2 📋 Task (VPL specification)

Input:

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

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

The following lines contain the matrix elements.

Output:

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

1.19.11.3 📌 Examples

Input	Output	Observation

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

```
%%writefile EP01_08.cpp
// your solution
```

```
Overwriting EP01_08.cpp
```

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

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

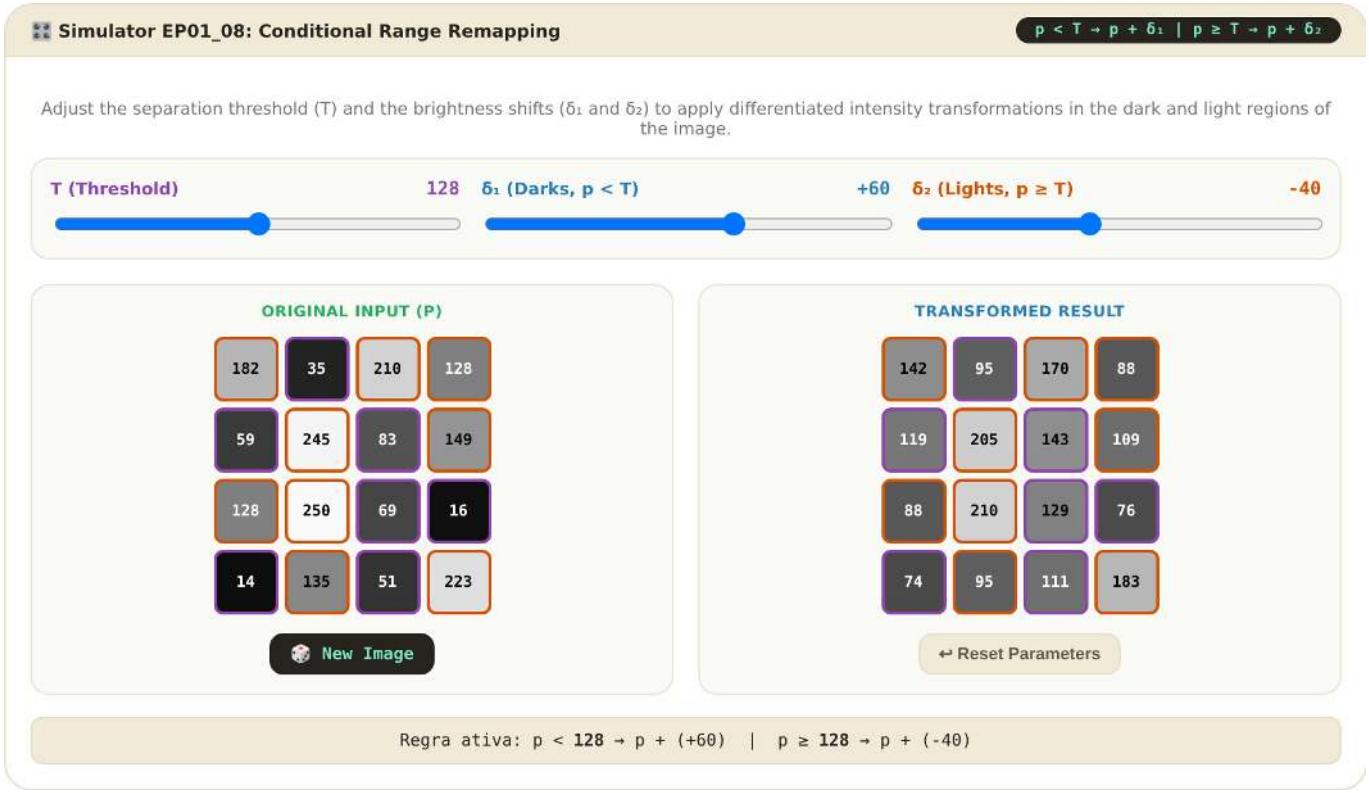
1.19.12 EP01_09 🏁 Checkerboard Pattern: Chess Matrix

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

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

📌 **Important**:

- The colors must alternate correctly both horizontally and vertically.
- The output must be the matrix printed line by line, with the elements separated by a space.



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

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

- See an interactive simulator for this problem at **Figure 1.19** (adjust the dimensions to visualize the construction of the grid and the corresponding textual output).

1.19.12.1 🧠 Synthetic Patterns

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

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

1.19.12.2 📋 Task (specification for VPL)

Input:

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

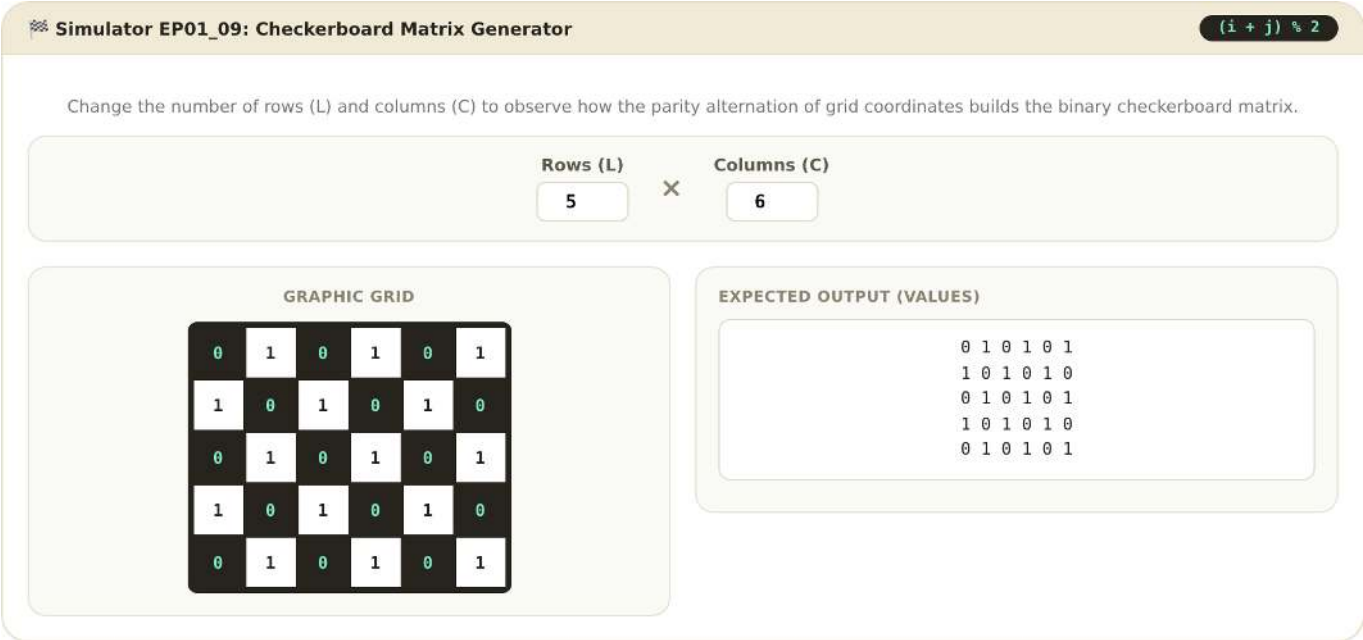
A line containing the integer `C` (columns).

Output:

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

1.19.12.3 📌 Examples

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



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

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

```
%%writefile EP01_09.cpp
// your solution
```

```
Overwriting EP01_09.cpp
```

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

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

1.19.13 EP01_10 📄 Metadata: Reading a PGM File

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

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

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

- After the header, the pixel data follows.

📌 **Important**:

- **File Reading:** You must open the file indicated in the example using Python's `open()` function.

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

1.19.13.1 🧠 Understanding the PGM Format

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

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

1.19.13.2 📋 Task (VPL specification)

Input:

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

Output:

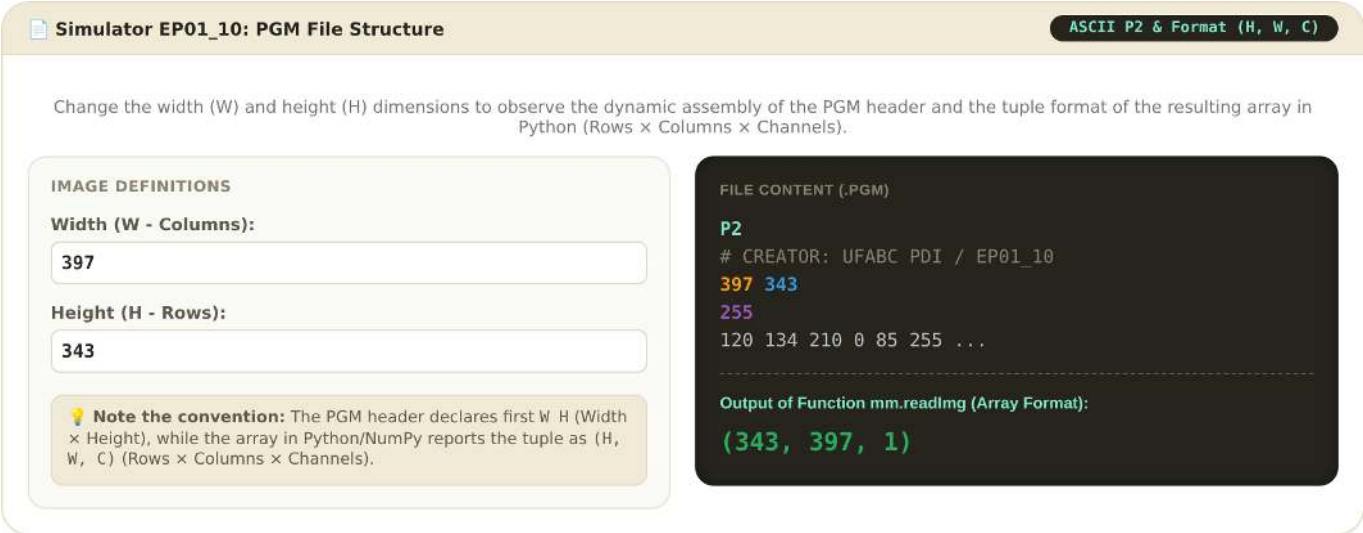
A tuple containing `(Height, Width, 1)`.

1.19.13.3 📌 Examples

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

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

```
%%writefile tmp/mm_out_2.cpp
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <filesystem>

// Compile: g++ -std=c++17 download.cpp -o download

namespace fs = std::filesystem;

bool downloadFile(const std::string& url, const std::string& filename) {
    // Use curl command to download the file
    std::string command = "curl -s -A 'Mozilla/5.0' -o \"" + filename + "\" \"" + url + "\"";
```



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

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

```
    int result = std::system(command.c_str());
    return result == 0;
}

int main() {
    const std::string BASE_URL =
"https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/all/cap01/imagens";
    const std::string file = "aula01fig03b.pgm";

    std::vector<std::string> files = {file};

    for (const auto& f : files) {
        if (!fs::exists(f)) {
            std::string url = BASE_URL + "/" + f;
            try {
                if (downloadFile(url, f)) {
                    std::cout << "  Arquivo baixado: " << f << std::endl;
                } else {
                    std::cout << "  Erro: Não encontrado em " << url << std::endl;
                }
            } catch (const std::exception& e) {
                std::cout << "  Erro: " << e.what() << " em " << url << std::endl;
            }
        }
    }

    return 0;
}
```

```
Overwriting tmp/mm_out_2.cpp
```

```
!g++ -I. -std=c++17 tmp/mm_out_2.cpp -o tmp/mm_out_2 \
  && ./tmp/mm_out_2
```

```
%%writefile EP01_10.cpp
// your solution
```

```
Overwriting EP01_10.cpp
```

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

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

1.19.14 EP01_11 ↗ Neighborhood Analysis: 1D Maximum Filter

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

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

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

📌 **Important**:

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

1.19.14.1 🧠 Why analyze neighbors?

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

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

1.19.14.2 📋 Task (VPL specification)

Input:

An integer `n`.

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

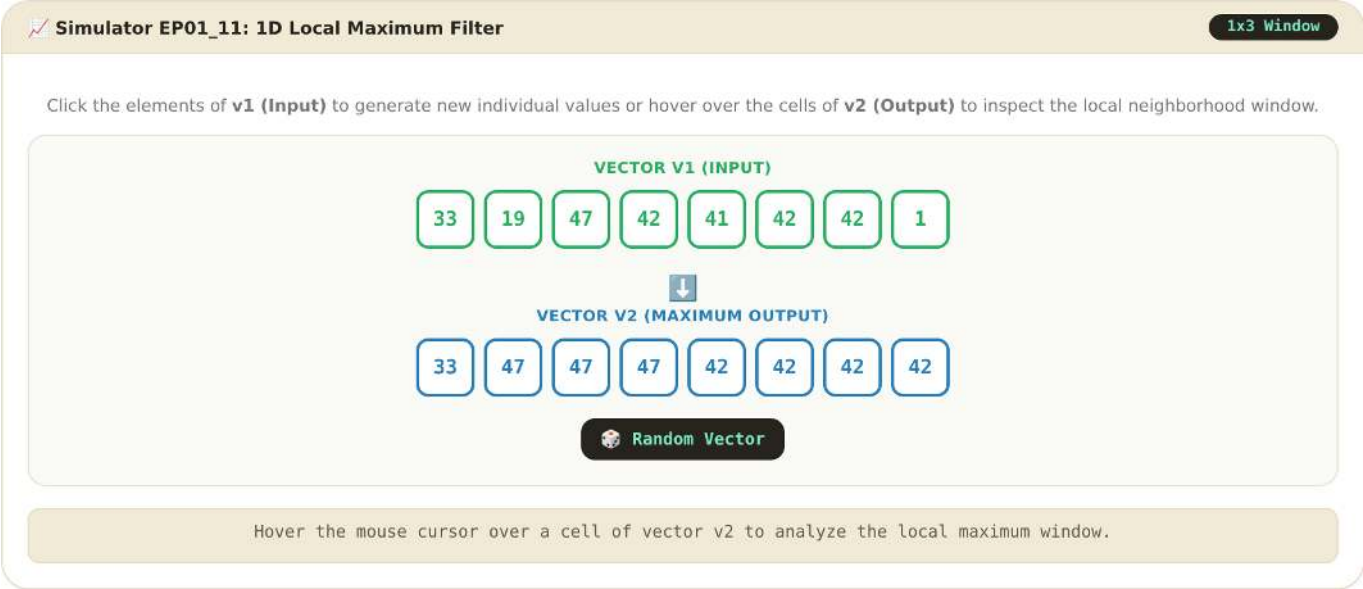
Output:

The string `v2:` on the first line.

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

1.19.14.3 📌 Examples

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



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

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

```
%%writefile EP01_11.cpp
// your solution
```

```
Overwriting EP01_11.cpp
```

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

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

Chapter 2

From Capture to Pixel - Sampling, Quantization, and Connectivity

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

2.1 Objectives

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

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

2.2 The Eye and the Camera - Elements of Visual Perception

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

2.2.1 Human vision

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

2.2.2 Digital sensors

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

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

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

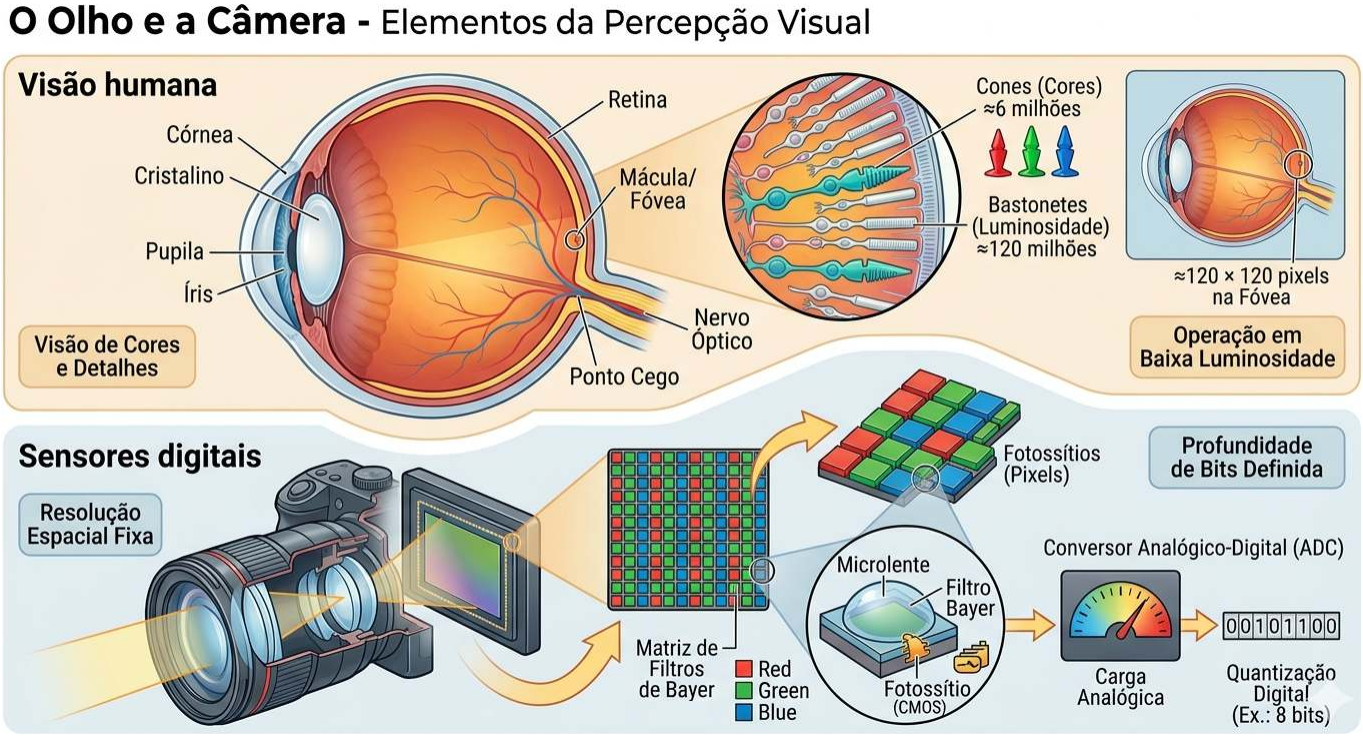


Figure 2.1: Didactic comparison between the biological and electronic visual systems: at the top, the anatomy of the human eye highlighting the retina and photoreceptors (cones and rods); below, the structure of a digital camera detailing the CMOS sensor, the Bayer filter array (RGGB), and the digital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

2.3 Optical Illusions: The Challenges of Visual Perception

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

2.3.1 Ambiguity and Context

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

2.3.2 Brightness and Local Contrast

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

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

2.3.3 Geometry and Perspective

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

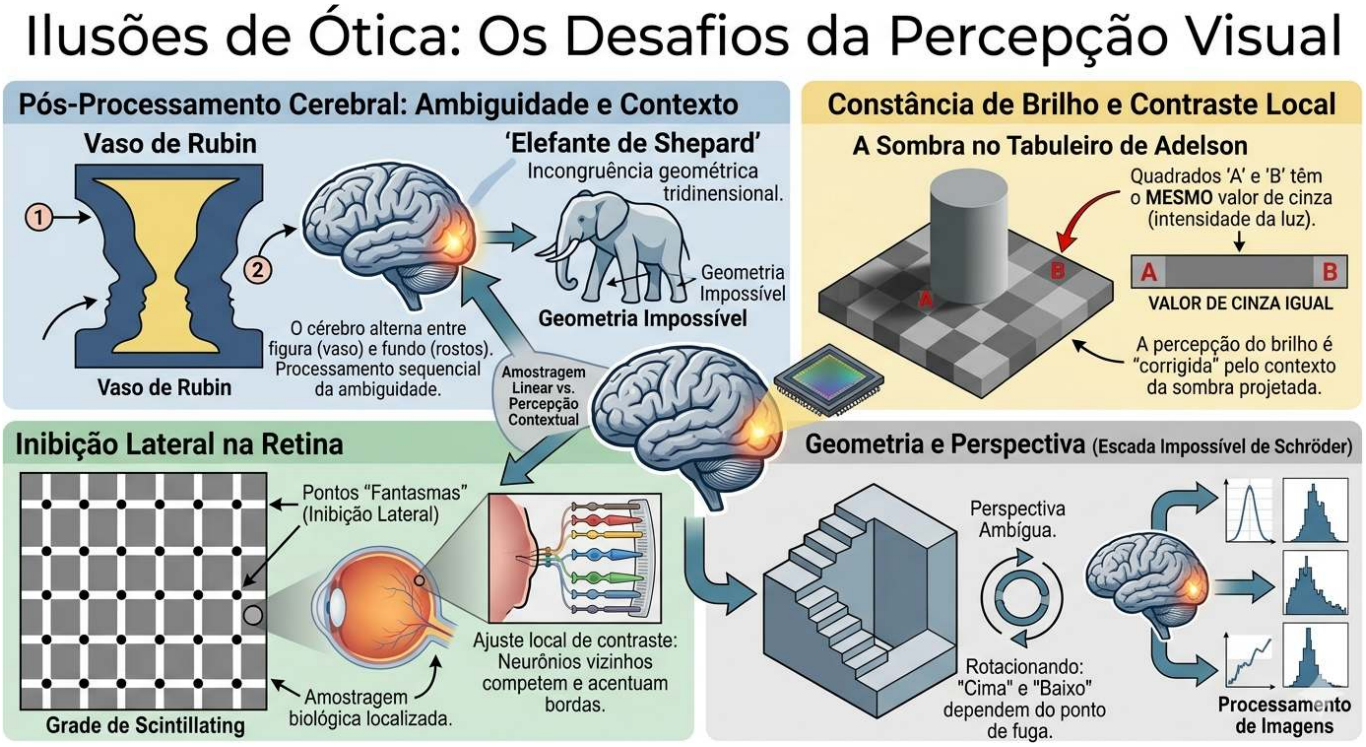


Figure 2.2: Collection of perceptual challenges: (top left) Rubin Vase — figure-ground ambiguity; (top center) Shepard Elephant — geometric incongruity; (top right) Adelson Shadow — brightness constancy based on context; (bottom left) Dot grid — lateral inhibition; (bottom right) Schröder Staircase.

2.4 The Mathematical Model of Image Formation

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

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

where:

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

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

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

2.4.1 Digital Representation and Spectral Channels

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

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

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

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

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

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

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

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

The conversion between intensity ranges is straightforward and given by:

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

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

Representation convention in `morph.hpp`: the library in this book adopts a single convention (Table 2.1), without the range and channel-order ambiguities that arise when combining multiple Python libraries.

Table 2.1: Image representation convention in morph.hpp — range, band order, no. of channels, and memory layout.

Aspect	`morph.hpp`
Sample type	`unsigned char` (`uint8`), range $[0, 255]$
Band order	**RGB** (via `stb_image`; no BGR inversion)
No. of channels	`1` (grayscale) or `3` (RGB)
Memory layout	$H \times W \times C$ interleaved (`data[(y*w + x)*channels + c]`)

The `Image` struct stores the pixels in a linear `std::vector<unsigned char>`, with the fields `h`, `w`, and `channels`. The `mm::read` function decodes with `stb_image` forcing 3 channels (or 1, when `grayscale=true`); therefore **any alpha channel in the file is discarded on reading**. To work in floating point, the same relation $f_{\text{norm}} = f/255$ holds.

In the following example cell, the RGBA version ($M \times N \times 4$) is assembled by stacking a synthetic alpha channel onto the read array — the notebook kernel is Python, and the display receives the `ndarray` in this $H \times W \times C$ layout.

2.4.2 Preparing the Practical Environment

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

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # C++ track build artifacts (.cpp, binary, PNGs)

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")

# The kernel is Python even in the C++ track: `mm` (morph.py) is used by the
# simulators, by the display of figures generated by the C++ binary, and by
# the mm::Image state between cells. cpp=True also downloads the compiled track
# (morph.hpp + stb_image*.h), used in the #include of %%writefile *.cpp cells.
import config
config.setup(demo=True, cpp=True)
from morph import mm
```

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

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

2.4.3 Implementation of Image Reading in `morph.hpp`

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

```
# morph.hpp is header-only; below, the body of the function mm::read().
import re, pathlib
hpp = pathlib.Path("morph.hpp").read_text()
m = re.search(r"inline Image read\(.*?\n\}", hpp, re.S)
print(m.group(0) if m else "(mm::read não encontrada em morph.hpp)")
```

```
inline Image read(const std::string& path_or_url, bool grayscale = false) {
    std::string local_path = path_or_url;
    bool is_url = path_or_url.rfind("http://", 0) == 0 ||
                  path_or_url.rfind("https://", 0) == 0;
    if (is_url) {
        local_path = "_mm_download_tmp.img";
        if (!_download(path_or_url, local_path))
            throw std::runtime_error("mm::read: falha ao baixar '" + path_or_url + "'");
    }

    int w, h, ch;
    int desired = grayscale ? 1 : 3;
    unsigned char* data = stbi_load(local_path.c_str(), &w, &h, &ch, desired);
    if (!data) {
        Image fallback;
        if (_try_read_ascii_pgm(local_path, fallback)) return fallback;
```

```
        throw std::runtime_error("mm::read: falha ao decodificar '" + path_or_url + "'");
    }

    Image img(h, w, desired);
    std::copy(data, data + (size_t)w * h * desired, img.data.begin());
    stbi_image_free(data);
    return img;
}
```

2.4.4 RGB and RGBA: Practical Example with the Mandrill Image

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

```
%%writefile tmp/fig_02_rgb_rgba.cpp
#define MM_OUT "tmp/fig_02_rgb_rgba.png"
//| label: fig-02-rgb-rgba
//| fig-cap: "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](https://sipi.usc.edu/database/database.php?volume=misc)
(domínio público)."
//| echo: true

#include "morph.hpp"
#include <iostream>
#include <filesystem>

int main() {
    // https://commons.wikimedia.org/wiki/File:Mandrill-k-means.png
    mm::Image img_rgb =
mm::read("https://upload.wikimedia.org/wikipedia/commons/a/ab/Mandrill-k-means.png");  // (M,
N, 3), uint8

    // Canal alfa: gradiente horizontal 0 -> 255 (esquerda -> direita)
    int h = img_rgb.h, w = img_rgb.w;
    mm::Image alpha(h, w);
    for (int x = 0; x < w; x++) {
        for (int y = 0; y < h; y++) {
            alpha.at(y, x) = static_cast<unsigned char>(255.0 * x / (w - 1));
        }
    }

    // Empilha RGB + alfa -> RGBA (M, N, 4)
    mm::Image img_rgba(h, w, 4);
    for (int y = 0; y < h; y++) {
        for (int x = 0; x < w; x++) {
            for (int c = 0; c < 3; c++) {
                img_rgba.at(y, x, c) = img_rgb.at(y, x, c);
            }
            img_rgba.at(y, x, 3) = alpha.at(y, x);
        }
    }

    std::cout << "RGB   -> shape=" << h << " x " << w << " x 3\n";
    std::cout << "RGBA  -> shape=" << h << " x " << w << " x 4\n";

    mm::show(std::vector<mm::Image>{img_rgb, img_rgba},
             MM_OUT,
             std::vector<std::string>{"RGB  -  M x N x 3", "RGBA  -  M x N x 4"}, 2);
```

```
// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_rgb, "tmp/fig_02_rgb_rgba_0.png");
mm::write(img_rgba, "tmp/fig_02_rgb_rgba_1.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_02_rgb_rgba.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_02_rgb_rgba.cpp -o tmp/fig_02_rgb_rgba \
  && ./tmp/fig_02_rgb_rgba \
  && test -f "tmp/fig_02_rgb_rgba.png" \
  || echo "  mm::show não gravou tmp/fig_02_rgb_rgba.png"
```

```
RGB   -> shape=512 x 512 x 3
RGBA  -> shape=512 x 512 x 4
[1] RGB  -  M x N x 3
[2] RGBA  -  M x N x 4
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_02_rgb_rgba_0.png"),
            mm.read("tmp/fig_02_rgb_rgba_1.png"),
        ],
        titles=[
            'RGB  -  M x N x 3',
            'RGBA  -  M x N x 4',
        ],
        cols=2,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_02_rgb_rgba_0.png
    (ver a versao Python)")
```

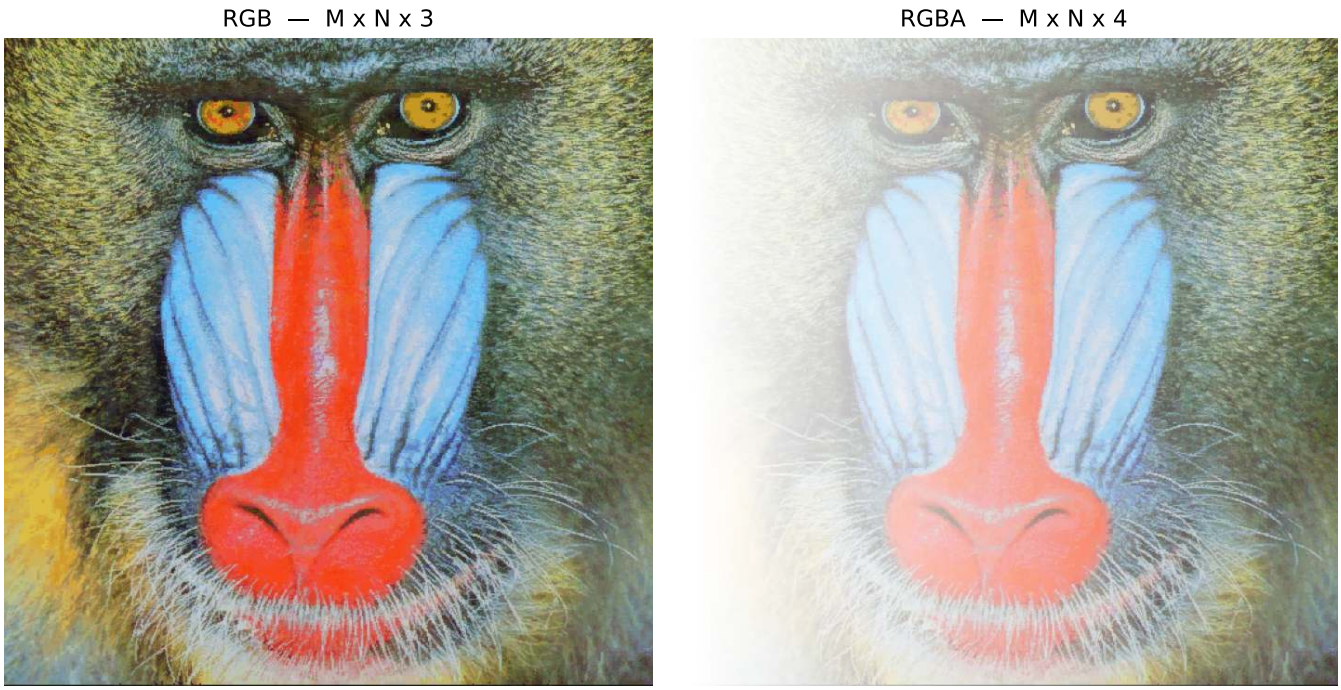


Figure 2.3: Imagem RGB (3 canais, $M \times N \times 3$) e versão RGBA (4 canais, $M \times N \times 4$) com transparência alfa gradual da esquerda para a direita. Imagem Mandrill — USC SIPI Image Database (domínio público).

2.5 Digitization: Sampling and Quantization

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

2.5.1 Sampling - Discretization of Space

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

2.5.2 Quantization - Discretization of Intensity

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

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

⚠ Quantization Error

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

2.5.3 Effects of Sampling and Quantization

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

```
%%writefile tmp/mm_out_1.cpp
#include "morph.hpp"
#include <iostream>
#include <filesystem>

int main() {
    // Imagem de exemplo (barbudo-rajado) - base das figuras de amostragem,
    // quantização e transformações geométricas deste capítulo.
    mm::Image img_color =
mm::read("https://upload.wikimedia.org/wikipedia/commons/c/c5/Area_de_Prote%C3%A7%C3%A3o_Ambiental_Quilo

    mm::Image img_gray0 = mm::gray(img_color);
    mm::Image img_gray = mm::crop(img_gray0, 820, 1850, 890, 1550);  // recorte p/ ver
detalhes
    std::cout << "Imagem original: " << img_color.h << "x" << img_color.w << "\n";


// [pdi:state-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp/state");
mm::write(img_gray, "tmp/state/img_gray_22.png");
// [pdi:state-io:end]
return 0;
}
```

```
Overwriting tmp/mm_out_1.cpp
```

```
!g++ -I. -std=c++17 tmp/mm_out_1.cpp -o tmp/mm_out_1 \
  && ./tmp/mm_out_1
```

```
Imagem original: 3067x2047
```

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

```
%%writefile tmp/fig_02_subamostragem.cpp
#define MM_OUT "tmp/fig_02_subamostragem.png"
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <cmath>

// #|
// #| label: fig-02-subamostragem
// #| fig-cap: "Efeito da subamostragem. Os títulos exibem as dimensões (W x H) e o tamanho da
matriz em memória (KB)."
// #| echo: true
// #| output: true

struct SubsampleResult {
    mm::Image res;
    std::string label;
};
```

```
SubsampleResult subsample_simple(const mm::Image& image, int f) {
    // Subsampling via slicing
    mm::Image reduced = mm::subsample(image, f);

    // Memory calculation in KB
    double mem_kb = (reduced.h * reduced.w * reduced.channels) / 1024.0;
    std::string label = std::to_string(reduced.w) + "x" + std::to_string(reduced.h) + ", " +
                        std::to_string(static_cast<int>(mem_kb)) + " KB\n(Factor " +
std::to_string(f) + ")";

    // Restore size for visualization (original H, W)
    mm::Image res = mm::resize(reduced, image.w, image.h, "nearest");
    return {res, label};
}

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_22.png");
// [pdi:state-io:end]

    std::vector<int> factors = {1, 4, 8, 12};

    // Generate results and separate into lists for mm::show
    std::vector<mm::Image> imgs_list;
    std::vector<std::string> titles_list;
    for (int f : factors) {
        SubsampleResult r = subsample_simple(img_gray, f);
        imgs_list.push_back(r.res);
        titles_list.push_back(r.label);
    }

    mm::show(imgs_list, MM_OUT, titles_list, 4);

    return 0;
}
```

```
Overwriting tmp/fig_02_subamostragem.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_02_subamostragem.cpp -o tmp/fig_02_subamostragem \
  && ./tmp/fig_02_subamostragem \
  && test -f "tmp/fig_02_subamostragem.png" \
  || echo "  mm::show não gravou tmp/fig_02_subamostragem.png"
```

```
[1] 660x1030, 663 KB
(Factor 1)
[2] 165x258, 41 KB
(Factor 4)
[3] 83x129, 10 KB
(Factor 8)
[4] 55x86, 4 KB
(Factor 12)
```

```
try:
    mm.show(mm.read("tmp/fig_02_subamostragem.png"), figsize=(16, 12))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_02_subamostragem.png (ver a versao Python)")
```

Figure 2.4: Efeito da subamostragem. Os títulos exibem as dimensões (W x H) e o tamanho da matriz em memória (KB).

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

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

```
%%writefile tmp/fig_02_quantizacao.cpp
#define MM_OUT "tmp/fig_02_quantizacao.png"
//| label: fig-02-quantizacao
//| fig-cap: "Efeito da redução da profundidade de bits. Os títulos exibem a quantidade de
bits, níveis e o tamanho em memória (KB)."
//| echo: true
//| output: true

#include "morph.hpp"
#include <vector>
#include <string>
#include <cmath>
#include <iostream>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_22.png");
// [pdi:state-io:end]

    // img_gray is already defined as mm::Image

    // Function to quantize and calculate metadata
    auto quantizeSimple = [](const mm::Image& image, int bits) {
        // levels = 2^bits
        int levels = (int)std::pow(2, bits);

        // Quantized image
        mm::Image quantized(image.h, image.w, image.channels);

        // Normalization and uniform quantization: floor(image/256 * levels)/levels * 255
        for (int y = 0; y < image.h; y++) {
            for (int x = 0; x < image.w; x++) {
                for (int c = 0; c < image.channels; c++) {
                    double val = (double)image.at(y, x, c) / 256 * levels;
                    double qval = std::floor(val) / levels * 255;
```

```
                    quantized.at(y, x, c) = (unsigned char)std::min(255, std::max(0,
(int)std::round(qval)));
                }
            }
        }

        // Memory calculation in KB
        double mem_kb = (double)(quantized.h * quantized.w * quantized.channels) / 1024;
        std::string label = std::to_string(bits) + " bits (" + std::to_string(levels) + "
níveis)\n"
                          + std::to_string((int)mem_kb) + " KB";

        return std::make_pair(quantized, label);
    };

    // List of bits for testing
    std::vector<int> bits_test = {8, 4, 2, 1};

    // Results containers
    std::vector<mm::Image> imgs_q;
    std::vector<std::string> titles_q;

    // Generate results
    for (int b : bits_test) {
        auto result = quantizeSimple(img_gray, b);
        imgs_q.push_back(result.first);
        titles_q.push_back(result.second);
    }

    mm::show(imgs_q, MM_OUT, titles_q, 4);

    return 0;
}
```

```
Overwriting tmp/fig_02_quantizacao.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_02_quantizacao.cpp -o tmp/fig_02_quantizacao \
  && ./tmp/fig_02_quantizacao \
  && test -f "tmp/fig_02_quantizacao.png" \
  || echo "  mm::show não gravou tmp/fig_02_quantizacao.png"
```

```
[1] 8 bits (256 níveis)
663 KB
[2] 4 bits (16 níveis)
663 KB
[3] 2 bits (4 níveis)
663 KB
[4] 1 bits (2 níveis)
663 KB
```

```
try:
    mm.show(mm.read("tmp/fig_02_quantizacao.png"), figsize=(16, 12))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_02_quantizacao.png
    (ver a versao Python)")
```

Figure 2.5: Efeito da redução da profundidade de bits. Os títulos exibem a quantidade de bits, níveis e o tamanho em memória (KB).

2.5.4 Technical Analysis

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

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

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

Table 2.2: Comparison between packing strategies and processing efficiency.

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

2.6 Relationships Between Pixels - Image Topology

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

2.6.1 Neighborhood

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

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

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

```
%%writefile tmp/fig_02_vizinhanca.cpp
#define MM_OUT "tmp/fig_02_vizinhanca.png"
#include "morph.hpp"
#include <iostream>

int main() {
    //| label: fig-02-vizinhanca
    //| fig-cap: "Ilustração de vizinhanças 4 em uma matriz 3x3. No centro (1,1), o pixel de
    interesse."
    //| echo: true
    //| output: true

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

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

    // or simply (test with numbers as arguments):
    mm::Image viz = mm::secross();

    // Display of the matrix for coordinate analysis
    mm::drawImgPlt(viz, MM_OUT);

    return 0;
}
```

```
Overwriting tmp/fig_02_vizinhanca.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_02_vizinhanca.cpp -o tmp/fig_02_vizinhanca \
  && ./tmp/fig_02_vizinhanca \
  && test -f "tmp/fig_02_vizinhanca.png" \
  || echo "  mm::show não gravou tmp/fig_02_vizinhanca.png"
```

```
0 1 0
1 1 1
0 1 0
```

```
try:
    mm.show(mm.read("tmp/fig_02_vizinhanca.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_02_vizinhanca.png
    (ver a versao Python)")
```

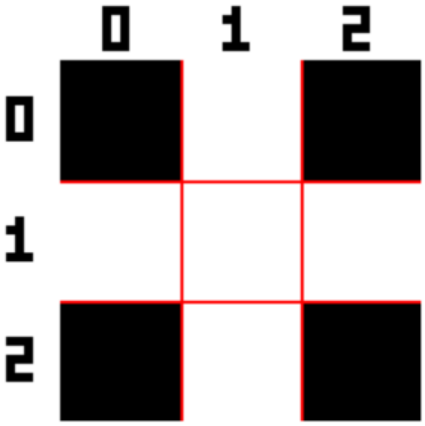


Figure 2.6: Ilustração de vizinhanças 4 em uma matriz 3x3. No centro (1,1), o pixel de interesse.

2.6.2 Adjacency, connectivity, and paths

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

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

2.6.3 Distances Between Pixels

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

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

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

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

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

i Practical Example

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

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

2.7 Image Storage

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

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

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

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

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

2.7.1 Example: Metadata Extraction and GPS Location

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

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

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

i C++ Track: Why EXIF Extraction Remains in Python

Metadata extraction is **container parsing** — decoding the EXIF header embedded in the file — rather than pixel processing: it is orthogonal to the sampling and quantization pipeline. `morph.hpp` decodes images with `stb_image`, which returns only the pixel matrix and discards the EXIF header. Since the kernel for this notebook is Python, even in the C++ track, this example uses Pillow-mode reading (`pil=True`) in both tracks. In a standalone C++ program, the equivalent path would involve vendoring a dedicated library: `easyexif` (read-only EXIF/GPS for JPEG, single header, zero dependencies) or `exiv2` (read **and** write, including IPTC/XMP).

```
%%writefile tmp/fig_01_natureza.cpp
#define MM_OUT "tmp/fig_01_natureza.png"
// Compile: g++ -std=c++17 -o program program.cpp -lmorph

#include "morph.hpp"
#include <iostream>
#include <string>
#include <vector>
#include <cmath>
#include <filesystem>

int main() {
    // | label: fig-01-natureza
    // | fig-cap: "Area de Proteção Ambiental Quilombos do Médio Ribeira - Barbudo-rajado
    (Malacoptila striata). Crédito: Thomas Fuhrmann (CC BY-SA 4.0)."
    // | echo: true

    std::string base = "https://upload.wikimedia.org/wikipedia/commons";
    std::string arquivo =
        "Area_de_Prote%C3%A7%C3%A3o_Ambiental_"
        "Quilombos_do_M%C3%A9dio_Ribeira_-_Thomas-"
        "Fuhrmann_%282023-_02%29_Malacoptila_striata.jpg";
    std::string url = base + "/c/c5/" + arquivo;
    std::string caminho = "imagens/barbudo-rajado.jpg";

    // 1. Reading - preserves PIL object with EXIF
    mm::Image img_obj;
    if (!std::filesystem::exists(caminho)) {
        std::filesystem::create_directories("imagens");
        img_obj = mm::read(url);
        mm::write(img_obj, caminho);      // saves preserving EXIF
    } else {
        img_obj = mm::read(caminho);
    }

    // Note: mm::Image doesn't preserve EXIF metadata
    // GPS extraction from EXIF is not available in this library

    // 2. Type diagnosis, dimensions and pixel access
    std::cout << "\nTipo PIL  : Image | Dimensões (x,y): " << img_obj.w << ", " << img_obj.h
<< std::endl;
    std::cout << "Pillow (0,0): (" << (int)img_obj.at(0, 0, 0) << ", " << (int)img_obj.at(0,
0, 1) << ", " << (int)img_obj.at(0, 0, 2) << ")" << std::endl;
    std::cout << "Tipo NumPy: Image | Dimensões [y,x,c]: " << img_obj.h << ", " << img_obj.w
<< ", " << img_obj.channels << std::endl;
    std::cout << "NumPy [0,0]: (" << (int)img_obj.at(0, 0, 0) << ", " << (int)img_obj.at(0, 0,
1) << ", " << (int)img_obj.at(0, 0, 2) << ")" << std::endl;

    // 3. Display
```

```
    mm::show(img_obj, MM_OUT);

    return 0;
}
```

```
Overwriting tmp/fig_01_natureza.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_01_natureza.cpp -o tmp/fig_01_natureza \
  && ./tmp/fig_01_natureza \
  && test -f "tmp/fig_01_natureza.png" \
  || echo "  mm::show não gravou tmp/fig_01_natureza.png"
```

```
Tipo PIL  : Image | Dimensões (x,y): 2047, 3067
Pillow (0,0): (58, 96, 0)
Tipo NumPy: Image | Dimensões [y,x,c]: 3067, 2047, 3
NumPy [0,0]: (58, 96, 0)
```

```
try:
    mm.show(mm.read("tmp/fig_01_natureza.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_01_natureza.png
    (ver a versao Python)")
```

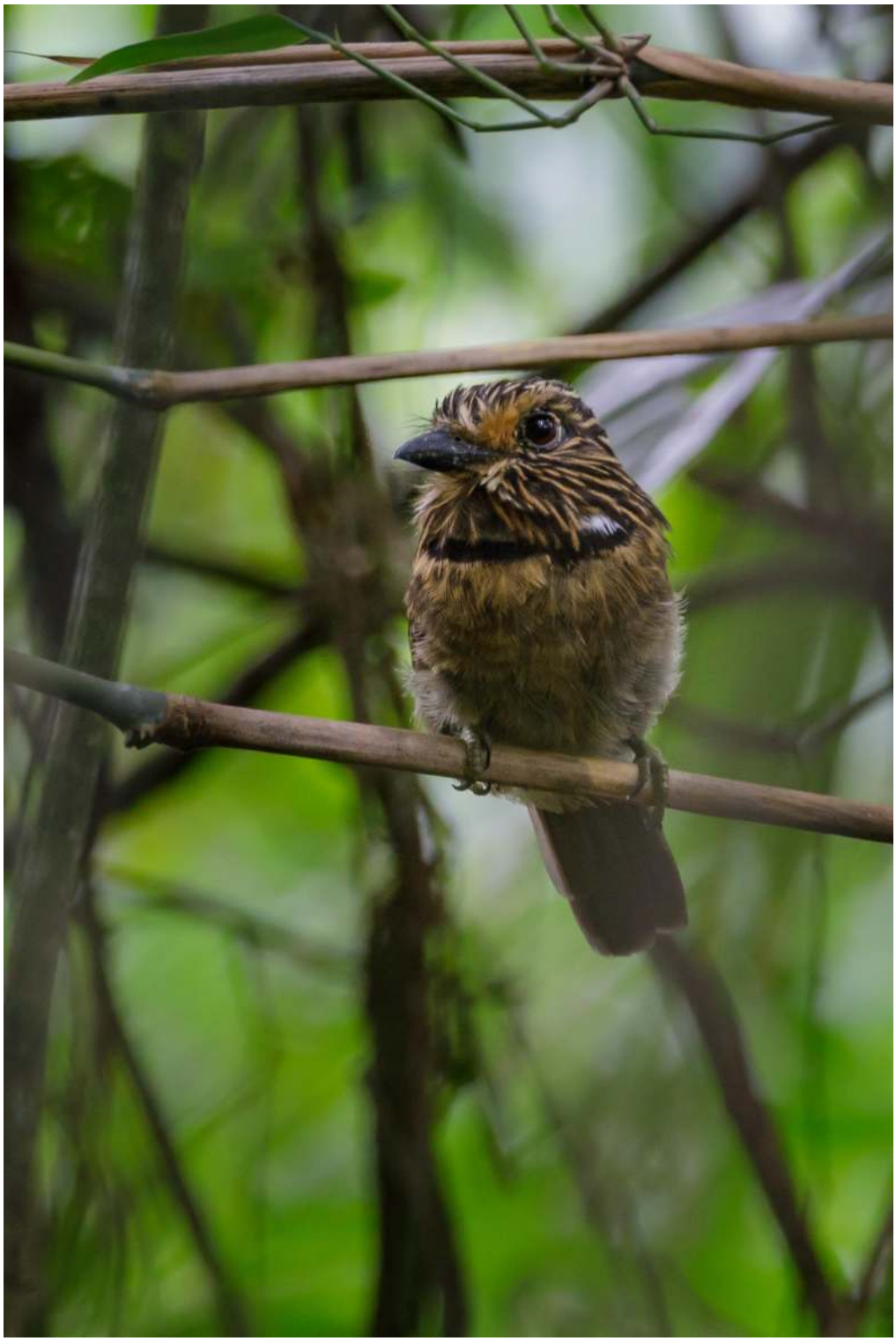


Figure 2.7: Area de Proteção Ambiental Quilombos do Médio Ribeira - Barbudo-rajado (Malacoptila striata). Crédito: Thomas Fuhrmann (CC BY-SA 4.0).

i Pedagogical Note: The Subtle Difference in Dimensions

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

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

`Columns/Width, Channels)` — in the example: `(3067, 2047, 3)`.
This distinction is fundamental to avoid indexing errors when implementing manual filters. While the Pillow object carries the **"where"** and **"when"** (context), the NumPy array carries the **"how much"** light (intensity) at each point of the image.

2.7.2 Why Is This Separation Important?

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

This class keeps the file "open" to allow access to the capture context before the data are converted into a raw matrix. Note that the pixel at `(0, 0)` is identical in both representations — `(58, 96, 0)` in Pillow and `[58, 96, 0]` in NumPy — confirming that both describe the same data, only with different interfaces. This separation is vital: pixels serve algorithms; metadata serve georeferencing, scientific cataloging, and corrections based on the acquisition hardware.

To inspect all EXIF metadata of a Pillow object:

```
from PIL import ExifTags

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

2.8 Basic Geometric Transformations

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

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

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

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

i Inverse transformation and interpolation

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

2.8.1 Translation

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

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

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

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

```
%%writefile tmp/fig_02_translacao.cpp
#define MM_OUT "tmp/fig_02_translacao.png"
//#| label: fig-02-translacao
//#| fig-cap: "Exemplo de translação da imagem do pássaro com deslocamentos (50,50) e
(100,50)."
//#| echo: true
//#| output: true
#include "morph.hpp"
#include <vector>
#include <string>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_22.png");
// [pdi:state-io:end]

    // img_gray is already initialized here

    mm::Image img_tx1 = mm::translate(img_gray, 50, 50);
    mm::Image img_tx2 = mm::translate(img_gray, 100, 50);
    mm::show(std::vector<mm::Image>{img_gray, img_tx1, img_tx2}, MM_OUT,
             std::vector<std::string>{"Original", "Translação (50,50)", "Translação
(100,50)"}, 3);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray, "tmp/fig_02_translacao_0.png");
mm::write(img_tx1, "tmp/fig_02_translacao_1.png");
mm::write(img_tx2, "tmp/fig_02_translacao_2.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_02_translacao.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_02_translacao.cpp -o tmp/fig_02_translacao \
  && ./tmp/fig_02_translacao \
  && test -f "tmp/fig_02_translacao.png" \
  || echo "  mm::show não gravou tmp/fig_02_translacao.png"
```

```
[1] Original
[2] Translação (50,50)
[3] Translação (100,50)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_02_translacao_0.png"),
            mm.read("tmp/fig_02_translacao_1.png"),
            mm.read("tmp/fig_02_translacao_2.png"),
        ],
        titles=[
            'Original',
            'Translação (50,50)',
            'Translação (100,50)',
        ],
        cols=3,
        figsize=(16, 12),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_02_translacao_0.png (ver a versao Python)")
```

Figure 2.8: Exemplo de translação da imagem do pássaro com deslocamentos (50,50) e (100,50).

2.8.2 Rotation

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

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

In the implementation, `cv2.getRotationMatrix2D` directly generates the first two rows of $\mathbf{T}_{\text{rot}}$ (a 2×3

matrix for `warpAffine`), also accepting a scale factor s that multiplies $\cos\theta$ and $\sin\theta$. See the example in Figure 2.9.

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

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

```
%%writefile tmp/fig_02_rotacao.cpp
#define MM_OUT "tmp/fig_02_rotacao.png"
//| label: fig-02-rotacao
//| fig-cap: "Exemplo de rotação da imagem do pássaro em 30° e 45° usando interpolação
bilinear."
//| echo: true
//| output: true

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_22.png");
// [pdi:state-io:end]

    // img_gray is already available (provided externally)

    mm::Image img_rot30 = mm::rotate(img_gray, 30, 1.0, "bilinear");
    mm::Image img_rot45 = mm::rotate(img_gray, 45, 1.0, "bilinear");

    mm::show(std::vector<mm::Image>{img_gray, img_rot30, img_rot45},
             MM_OUT,
             std::vector<std::string>{"Original", "Rotação 30°", "Rotação 45°"},
             3);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray, "tmp/fig_02_rotacao_0.png");
mm::write(img_rot30, "tmp/fig_02_rotacao_1.png");
mm::write(img_rot45, "tmp/fig_02_rotacao_2.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_02_rotacao.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_02_rotacao.cpp -o tmp/fig_02_rotacao \
  && ./tmp/fig_02_rotacao \
  && test -f "tmp/fig_02_rotacao.png" \
  || echo "  mm::show não gravou tmp/fig_02_rotacao.png"
```

```
[1] Original
[2] Rotação 30°
[3] Rotação 45°
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_02_rotacao_0.png"),
            mm.read("tmp/fig_02_rotacao_1.png"),
            mm.read("tmp/fig_02_rotacao_2.png"),
        ],
        titles=[
            'Original',
            'Rotação 30°',
            'Rotação 45°',
        ],
        cols=3,
        figsize=(16, 12),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_02_rotacao_0.png
    (ver a versao Python)")
```

Figure 2.9: Exemplo de rotação da imagem do pássaro em 30° e 45° usando interpolação bilinear.

2.8.3 Scaling (Resizing)

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

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

When $s > 1$ (enlargement), pixels of the destination image map to non-integer positions in the source — requiring interpolation to estimate the value. When $s < 1$ (reduction), multiple source pixels contribute

to a single destination pixel — requiring **decimation**. The same three methods described for rotation are available in `mm::resize`, with the difference that here the visual impact is more noticeable: in enlargement, `nearest` produces a blocky effect (*pixelation*), whereas `bicubic` better preserves edge sharpness, as shown in Table 2.5:

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

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

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

```
%%writefile tmp/fig_02_escala_detalhe.cpp
#define MM_OUT "tmp/fig_02_escala_detalhe.png"
#include "morph.hpp"
#include <iostream>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_22.png");
// [pdi:state-io:end]

    // 1. Crop of the beak region
    int y = 210, x = 40, offset = 40;
    mm::Image crop = mm::crop(img_gray, y - offset, y + offset, x - offset, x + offset);
    std::cout << "Imagem: " << img_gray.h << "x" << img_gray.w << " | Crop: " << crop.h << "x"
<< crop.w << "\n";

    // 2. Enlarge 4x with mm.resize
    mm::Image crop_nearest  = mm::resize(crop, (double)4, "nearest");
    mm::Image crop_bilinear = mm::resize(crop, (double)4, "bilinear");

    // 3. Comparative display
    mm::show(
        std::vector<mm::Image>{crop, crop_nearest, crop_bilinear},
        MM_OUT,
        std::vector<std::string>{"Original (recorte)", "Vizinho mais próximo (4x)", "Bilinear
(4x)"},
        3
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(crop, "tmp/fig_02_escala_detalhe_0.png");
mm::write(crop_nearest, "tmp/fig_02_escala_detalhe_1.png");
mm::write(crop_bilinear, "tmp/fig_02_escala_detalhe_2.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_02_escala_detalhe.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_02_escala_detalhe.cpp -o tmp/fig_02_escala_detalhe \
  && ./tmp/fig_02_escala_detalhe \
  && test -f "tmp/fig_02_escala_detalhe.png" \
  || echo "  mm::show não gravou tmp/fig_02_escala_detalhe.png"
```

```
Imagem: 1030x660 | Crop: 80x80
[1] Original (recorte)
[2] Vizinho mais próximo (4x)
[3] Bilinear (4x)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_02_escala_detalhe_0.png"),
            mm.read("tmp/fig_02_escala_detalhe_1.png"),
            mm.read("tmp/fig_02_escala_detalhe_2.png"),
        ],
        titles=[
            'Original (recorte)',
            'Vizinho mais próximo (4×)',
            'Bilinear (4×)',
        ],
        cols=3,
        figsize=(16, 12),
        dpi=200,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_02_escala_detalhe_0.png (ver a versao Python)")
```

Figure 2.10: Comparação de interpolação com zoom no detalhe do olho (recorte 60×60, ampliado 4×). Note o efeito de blocos no vizinho mais próximo vs. a suavização na bilinear.

2.8.4 Shearing

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

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

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

```
%%writefile tmp/fig_02_cisalhamento.cpp
#define MM_OUT "tmp/fig_02_cisalhamento.png"
//| label: fig-02-cisalhamento
//| fig-cap: "Exemplo de cisalhamento da imagem do pássaro: horizontal (shx=0.3), vertical
(shy=0.3) e combinado (shx=0.2, shy=0.2)."
//| echo: true
//| output: true

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_22.png");
// [pdi:state-io:end]

    mm::Image img_shx = mm::shear(img_gray, 0.3, 0.0, "bilinear");
    mm::Image img_shy = mm::shear(img_gray, 0.0, 0.3, "bilinear");
    mm::Image img_shc = mm::shear(img_gray, 0.2, 0.2, "bilinear");

    mm::show(
        std::vector<mm::Image>{img_gray, img_shx, img_shy, img_shc},
        MM_OUT,
        std::vector<std::string>{"Original", "Horiz. (shx=0.3)", "Vert. (shy=0.3)", "Combinado
(0.2, 0.2)"},
        4
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray, "tmp/fig_02_cisalhamento_0.png");
mm::write(img_shx, "tmp/fig_02_cisalhamento_1.png");
mm::write(img_shy, "tmp/fig_02_cisalhamento_2.png");
mm::write(img_shc, "tmp/fig_02_cisalhamento_3.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_02_cisalhamento.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_02_cisalhamento.cpp -o tmp/fig_02_cisalhamento \
  && ./tmp/fig_02_cisalhamento \
  && test -f "tmp/fig_02_cisalhamento.png" \
  || echo "  mm::show não gravou tmp/fig_02_cisalhamento.png"
```

```
[1] Original
[2] Horiz. (shx=0.3)
[3] Vert. (shy=0.3)
[4] Combinado (0.2, 0.2)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_02_cisalhamento_0.png"),
```

```
            mm.read("tmp/fig_02_cisalhamento_1.png"),
            mm.read("tmp/fig_02_cisalhamento_2.png"),
            mm.read("tmp/fig_02_cisalhamento_3.png"),
        ],
        titles=[
            'Original',
            'Horiz. (shx=0.3)',
            'Vert. (shy=0.3)',
            'Combinado (0.2, 0.2)',
        ],
        cols=4,
        figsize=(16, 12),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_02_cisalhamento_0.png (ver a versao Python)")
```

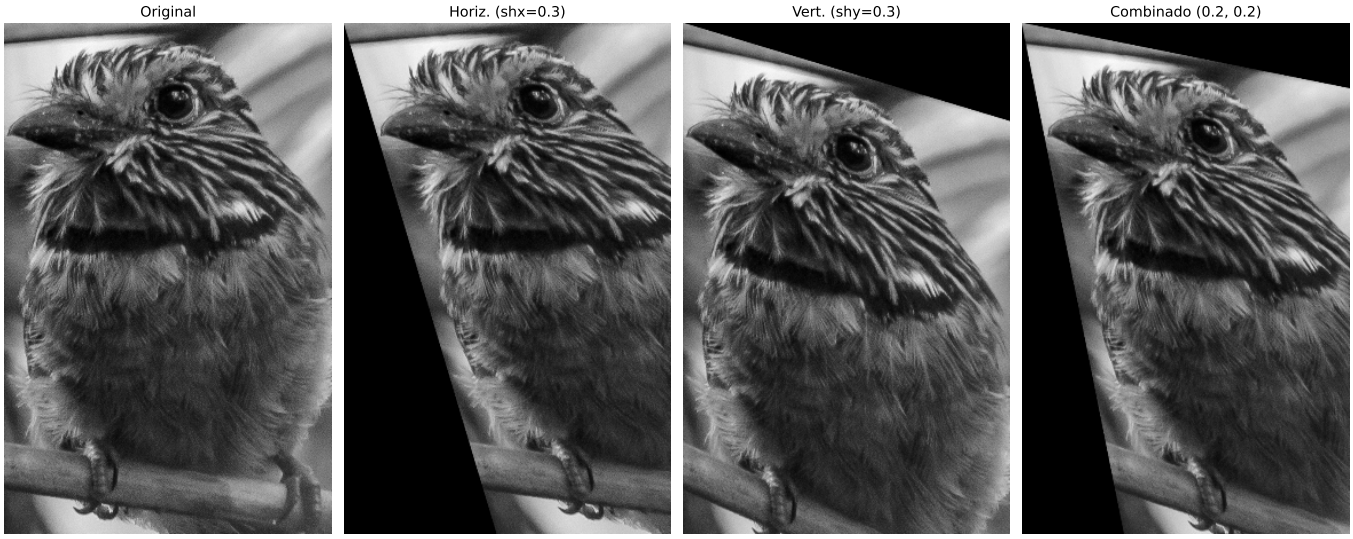


Figure 2.11: Exemplo de cisalhamento da imagem do pássaro: horizontal (shx=0.3), vertical (shy=0.3) e combinado (shx=0.2, shy=0.2).

2.9 Summary

This chapter presented the fundamentals of image digitization and topology:

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

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

2.10 🤖 Using Gemini Notebook as a Complementary Tutor

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

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

🎓 Study with the Intelligent Tutor

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

🌐 Language and Programming Language

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

⚠️ Notice on AI-Generated Content

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

2.11 Exercise List

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

Chapter References

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

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

2.12 Chapter 2 - Digital Image Processing: Exercises Proposed (EPs)

This chapter presents a series of proposed exercises (EPs) that complement the theoretical content discussed in the main textbook. These activities aim to consolidate the concepts of digital image representation, color models, and basic operations on pixels, with a focus on practical implementation in Python.

2.13 2.1 - Objectives

- To understand the fundamental concepts of digital images and their representation in computational environments.
- To manipulate images through the Python programming language, using libraries such as NumPy and OpenCV.
- To apply basic operations, including reading, displaying, converting, and saving images.
- To explore different color models and their applications in image processing.
- To develop computational thinking skills by solving problems within the scope of image processing.

2.14 2.2 - Proposed Exercises

For each of the proposed exercises, implement the requested functionality, validating your solution with the provided test cases.

2.14.1 EP 2.1 - Basic Image Reading

Write a program that reads an image from a file, displays it in a window, and saves a copy in another format. Utilize the libraries `cv2` and `matplotlib` as needed.

2.14.2 EP 2.2 - Conversion Between Color Models

Implement a function that converts an image from the RGB model to the HSV model without using the built-in conversion functions from specialized libraries. Compare your result with the output generated by OpenCV's `cvtColor` function.

2.14.3 EP 2.3 - Pixel Manipulation

Develop a script that inverts the colors of an image (negative effect), both on the entire image and on a specific region of interest (ROI). Display the original and processed images side by side.

2.14.4 EP 2.4 - Arithmetic Operations

Explore the effect of arithmetic operations (addition, subtraction, multiplication, and division) on image brightness and contrast. Apply these operations to two distinct images or to a constant value, and discuss the observed results.

2.15 2.3 - Additional Challenges

1. Implement a program that creates a color gradient image based on pixel coordinates.
2. Simulate the effect of the "sepia" filter by applying a linear transformation to the RGB channels.
3. Compare the performance of a pixel-by-pixel operation (using loops) with a vectorized implementation using NumPy.

2.16 2.4 - Submission and Evaluation Guidelines

- All code must be properly commented and organized into functions.
- The use of `numpy` and `opencv-python` is recommended. To install the dependencies, run:

```
!pip install numpy opencv-python matplotlib
```

- Test your solutions using the provided image datasets, available in the course's repository.
- The final notebook must be submitted via the associated Google Colab link at the beginning of this chapter.

This concludes the proposed exercises for Chapter 2. In the following chapter, the focus will shift to the study of spatial filtering techniques.

2.17 💻 Practical Part with Programming Exercises

🎯 Objective of this Notebook

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

Download

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

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # C++ track build artifacts (.cpp, binary, PNGs)

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")

# The kernel is Python even in the C++ track: `mm` (morph.py) is used by the
# simulators, by the display of figures that the C++ binary generates, and by the
# mm::Image state between cells. cpp=True also downloads the compiled track
# (morph.hpp + stb_image*.h), used in the #include of %%writefile *.cpp cells.
import config
config.setup(testsuite=True, cpp=True)
from morph import mm
from testsuite import TestSuite
```

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

Executing Tests

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

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

```
code = """
from morph import mm
```

```
# ... your code here ...
"""
TestSuite("EP04_01").run_code(code)
```

2.17.1 EP02_01 ⊙ Brightness and Contrast Adjustment

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

2.17.1.1 📋 Implementation Guidelines

The algorithm should follow the execution flow below:

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

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

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

2.17.1.2 📌 Computational Constraints

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

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

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

2.17.1.3 🧠 Theoretical Background

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

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

2.17.1.4 📦 Input and Output Specification (VPL)

Input:

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

Output:

- Transformed matrix structured into L rows and C columns.

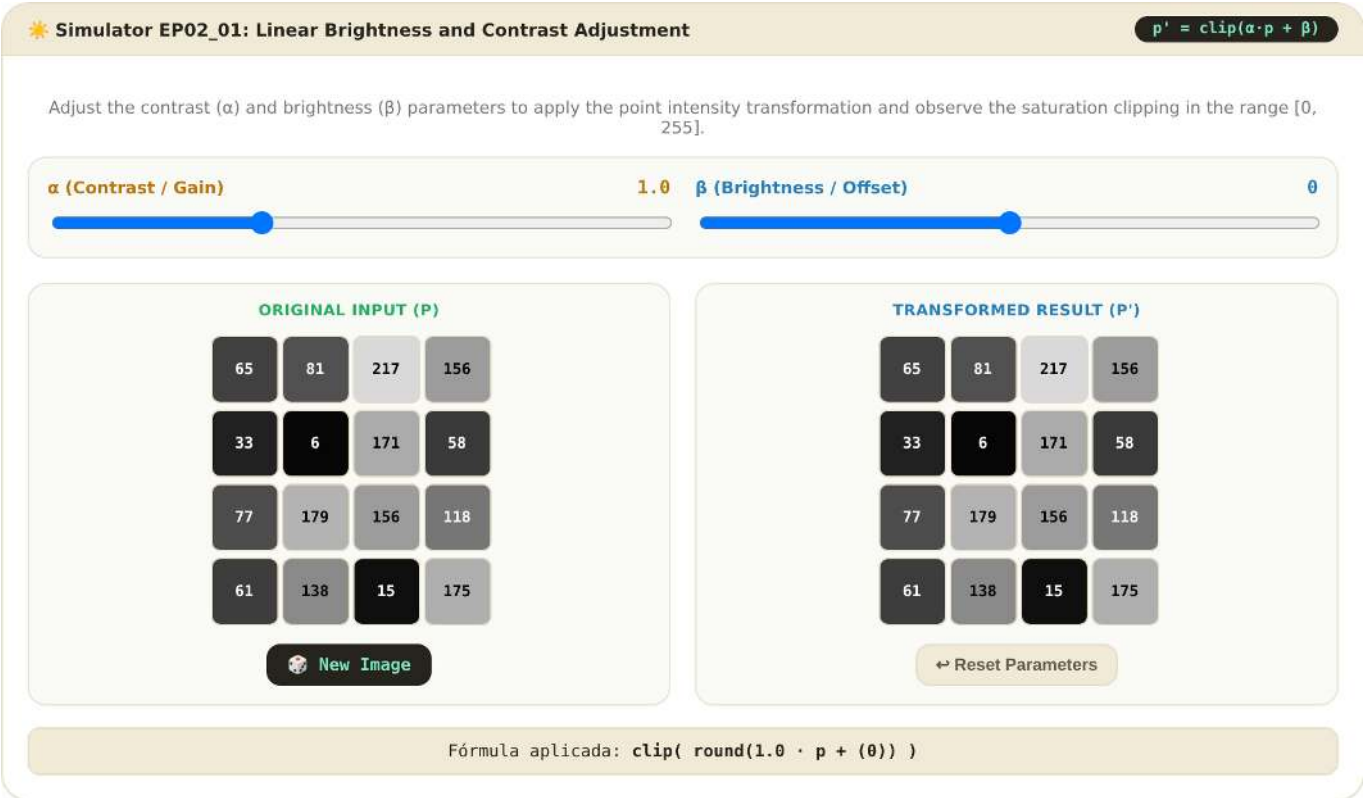
2.17.1.5 📌 Examples

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

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

Running the Tests

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



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

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

```
%%writefile EP02_01.cpp
// your solution
```

```
Overwriting EP02_01.cpp
```

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

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

2.17.2 EP02_02 🔬 Spatial Subsampling

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

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

📌 **Important**:

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

2.17.2.1 🧠 Discretization of Space

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

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

2.17.2.2 📋 Task (specification for VPL)

Input:

The first line contains `L`.

The second line contains `C`.

The third line contains the factor `f`.

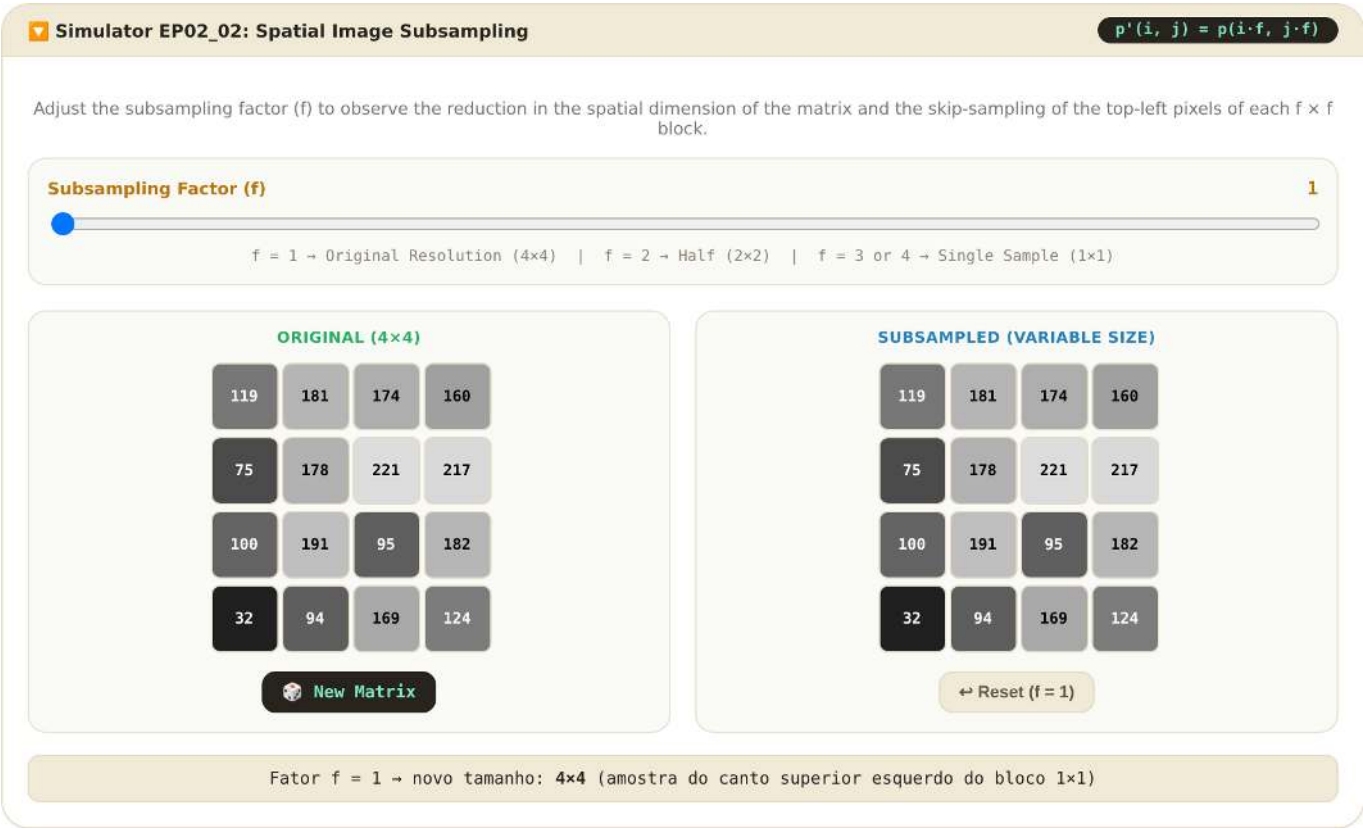
The following lines contain the elements of the $L \times C$ matrix.

Output:

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

2.17.2.3 📌 Examples

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



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

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

```
%%writefile EP02_02.cpp
// your solution
```

```
Overwriting EP02_02.cpp
```

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

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

2.17.3 EP02_03 🎨 Gray Level Quantization

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

- Read two integers **L** and **C**, representing the dimensions of the matrix.
- Read an integer k ($1 \leq k \leq 8$), representing the new number of bits of the image.
- Calculate the number of levels ($N = 2^k$) and the interval size (step).
- For each pixel p, calculate the new value p' by mapping it to the index of the corresponding discretized level (ranging from 0 to $2^k - 1$).

- Print the resulting matrix with the same original dimension values.
- See Figure 2.14 for a simulation of this EP.

📌 **Important**:

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

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

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

2.17.3.1 🧠 Amplitude Discretization

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

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

2.17.3.2 📋 Task (VPL specification)

Input:

The first line contains `L`.

The second line contains `C`.

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

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

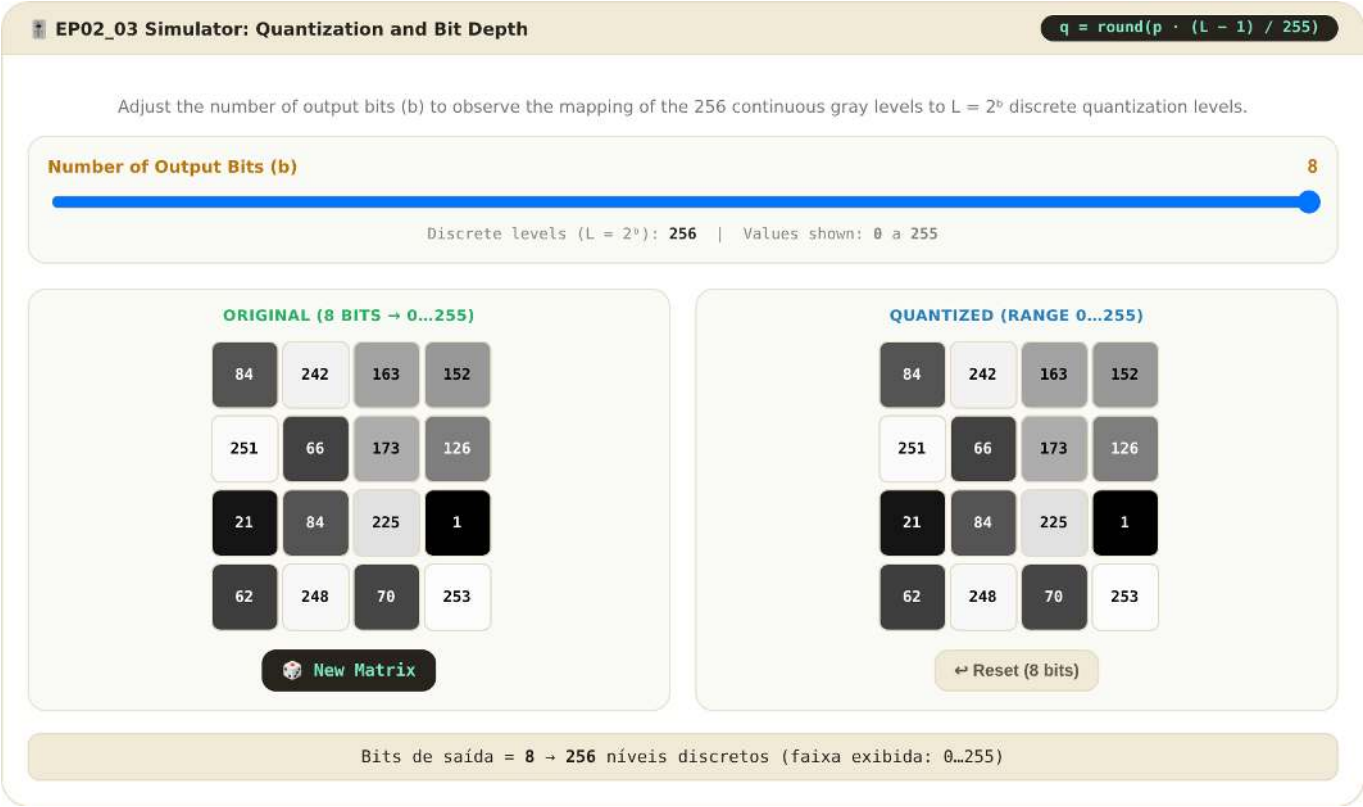
Output:

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

2.17.3.3 📌 Examples

Input	Output	Observation
1 4 2 0 80 170 255	0 1 2 3	With $k = 2$, we have $2^2 = 4$ discrete levels available $(0, 1, 2, 3)$. The step is $256/4 = 64$. Applying integer division element-wise: $0//64 = 0$, $80//64 = 1$, $170//64 = 2$, $255//64 = 3$.

Input	Output	Observation
1 5 1 10 50 120 200 250	0 0 0 1 1	With $k = 1$, we have $2^1 = 2$ levels (0 and 1). Step $= 256/2 = 128$. Pixels less than 128 result in 0, and pixels greater than or equal to 128 result in 1.



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

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

```
%%writefile EP02_03.cpp
// your solution
```

```
Overwriting EP02_03.cpp
```

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

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

2.17.4 EP02_04 📐 Distance Transform in Binary Images

Given a binary image where pixels with value **1** represent the *object* and pixels with **0** represent the *background*, the distance of a background pixel is the **smallest distance to the nearest object pixel**. Object pixels are assigned a distance of **0**. For simplicity, consider that the image has **only a single object consisting of one pixel with value 1**.

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

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

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

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

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

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

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

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

2.17.4.2 📌 Technical Requirements

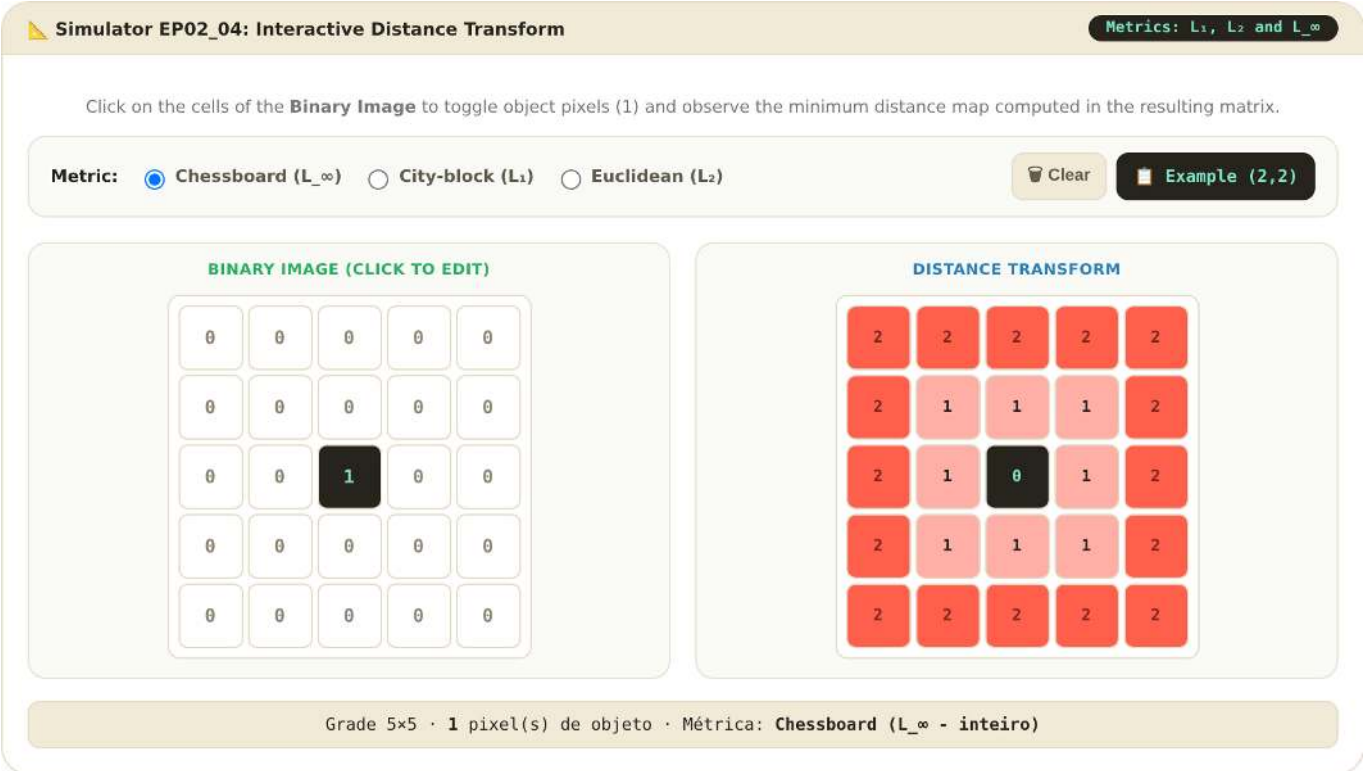
- **Input:** * First line: L and C (integers).
  - Second line: metric name (`euclidean`, `cityblock`, or `chessboard`).
  - Then, the binary matrix $L \times C$ (values 0 or 1).
- **Object pixels (1):** distance = 0 (or 0.00 for Euclidean).
- **Background pixels (0):** distance to the single object pixel in the image.
- **Rounding (Euclidean):** print with **2 decimal places** (format `:.2f`). City-block and Chessboard produce integers — print without decimals.
- **Output:** space-separated values, one row of the matrix per line.
- See Figure 2.15 for a simulation of this EP.

2.17.4.3 📌 Examples

Input	Output	Observation
4 4 chessboard 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0	2 2 2 2 2 1 1 1 2 1 0 1 2 1 1 1	The Chessboard distance is $\max(\|dx\|, \|dy\|)$. The only object pixel is $(2, 2) = 0$; the others store their minimum distance to it.

2.17.4.4 📌 Final Remarks

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



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

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

```
%%writefile EP02_04.cpp
// your solution
```

```
Overwriting EP02_04.cpp
```

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

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

2.17.5 EP02_05 Image Translation

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

- Read two integers **L** and **C**, representing the dimensions of the matrix.
- Read two integers t_x (horizontal displacement) and t_y (vertical displacement).
- Read the integer values of the original matrix.
- Calculate the new position (x', y') for each original pixel (x, y).
- Print the resulting matrix with the same dimensions as the original.
- See a simulation of this EP in Figure 2.16.

📌 **Important**:

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

2.17.5.1 🧠 Spatial Displacement

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

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

Which results in the simple equations:

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

2.17.5.2 📋 Task (specification for VPL)

Input:

The first line contains `L`.

The second line contains `C`.

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

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

Output:

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

2.17.5.3 📌 Examples

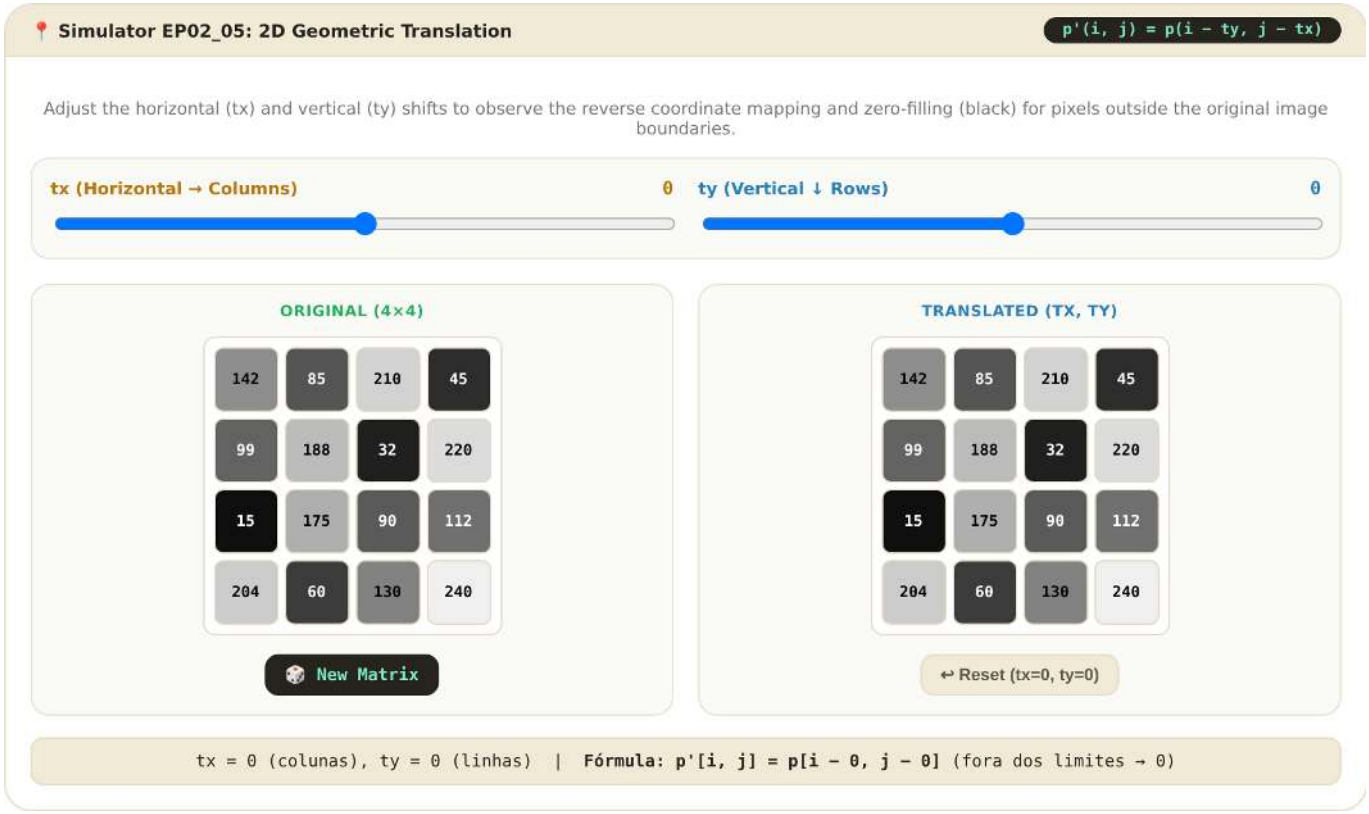
Input	Output	Observation
2 2 1 1 10 20 30 40	0 0 0 10	**Displacement** ($t_x = 1, t_y = 1$)**:** Each pixel moves one position to the right (horizontal) and one position down (vertical). Pixel $(0, 0) = 10$ moves to destination $(1, 1)$ (bottom-right corner). Empty positions are filled with 0.
3 3 -1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

```
%%writefile EP02_05.cpp
// your solution
```

```
Overwriting EP02_05.cpp
```

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

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



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

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

2.17.6 EP02_06 🔄 Image Rotation

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

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

📌 **Important**:

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

2.17.6.1 🧠 Angle Transformation

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

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

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

2.17.6.2 📋 Task (specification for VPL)

Input:

The first line contains `L`.

The second line contains `C`.

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

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

Output:

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

2.17.6.3 📌 Examples

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

```
%%writefile EP02_06.cpp
// your solution
```

```
Overwriting EP02_06.cpp
```

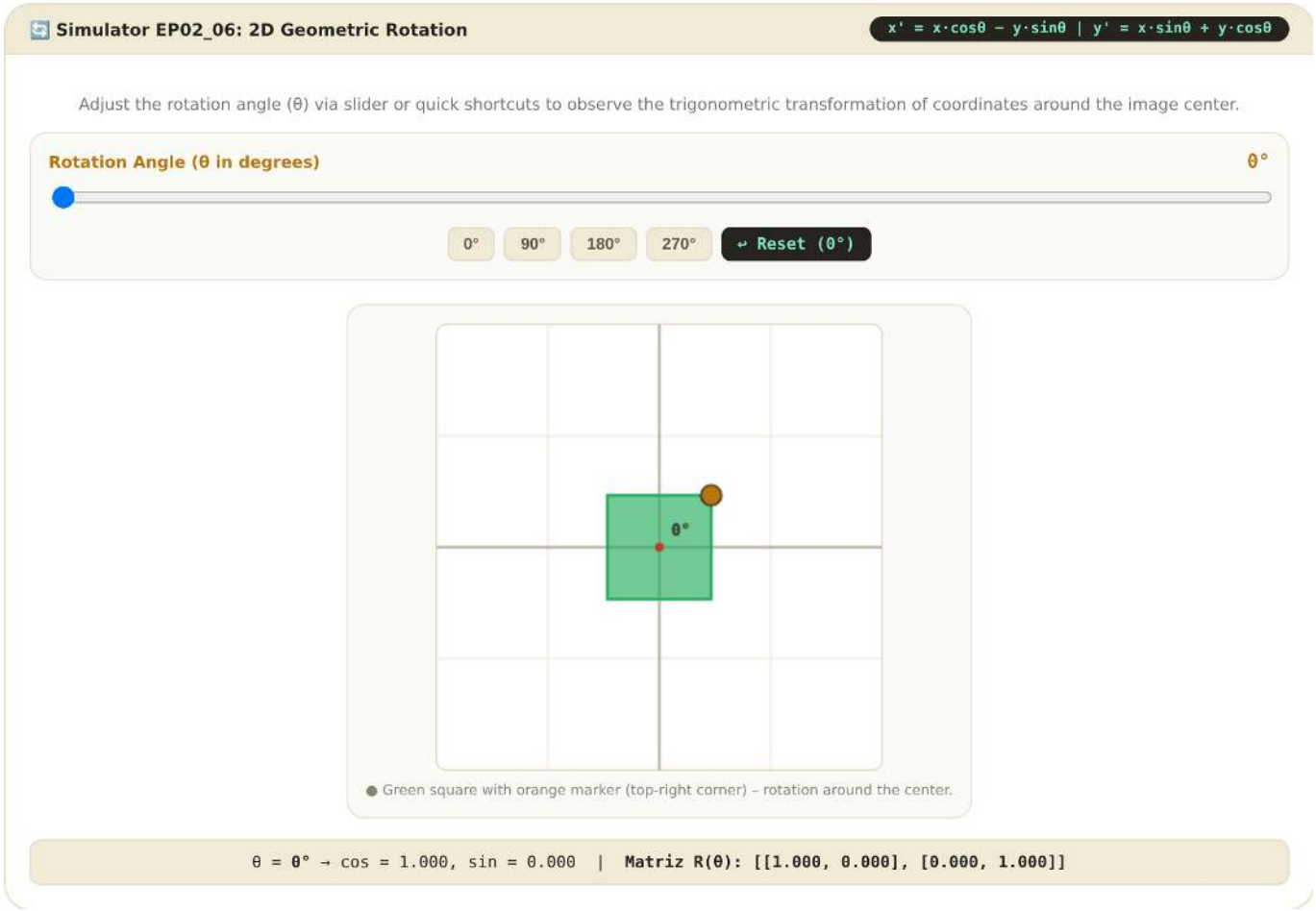
```
TestSuite("EP02_06.cpp").run()
```

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

2.17.7 EP02_07 🔍 Resizing (Scaling)

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

- Read two integers **L** and **C**, representing the dimensions of the original matrix.
- Read two real values s_x (scale along rows) and s_y (scale along columns).
- Read a *string* representing the interpolation **method** (`nearest` or `bilinear`).



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

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

- Read the integer values of the original matrix.
- Compute the new dimensions: $L' = \text{round}(L \times s_x)$ and $C' = \text{round}(C \times s_y)$.
- Print the resulting matrix with the new dimensions.
- See Figure 2.18 for a simulation of this EP.

📌 **Important**:

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

2.17.7.1 🧠 Interpolation for Enlargement/Reduction

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

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

2.17.7.2 📋 Task (specification for VPL)

Input:

The first line contains `L`.

The second line contains `C`.

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

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

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

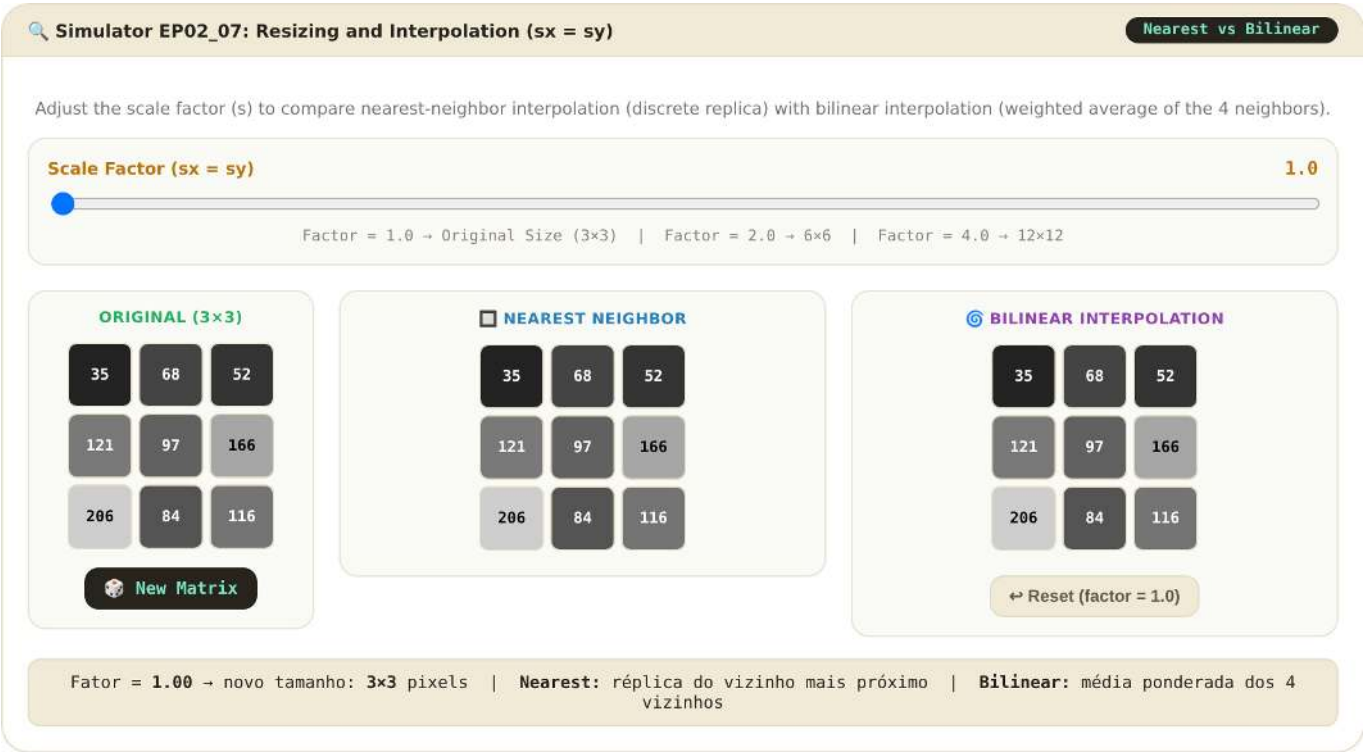
Output:

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

2.17.7.3 📌 Examples

Input	Output	Observation
2 2 2.0 2.0 nearest 1 2 3 4	1 1 2 2 1 1 2 2 3 3 4 4 3 3 4 4	2× enlargement: each original pixel is replicated in a 2×2 block. The 2×2 image becomes 4×4.

Input	Output	Observation
2 2 0.5 0.5 nearest 10 20 30 40	10	0.5× reduction: the 2×2 image becomes 1×1. With nearest, the only output pixel samples position $(0, 0) = 10$.



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

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

```
# Not yet ported to this language in this version - conceptual reference in Python.
%%writefile EP02_07.py
# Python code
import numpy as np
from morph import mm

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

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

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

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

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

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

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

2.17.8 EP02_08 🔀 Shear Transformation

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

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

📌 **Important**:

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

2.17.8.1 🧠 Affine Distortion

Shear alters the image geometry by tilting its axes. The relationship between the original coordinates (x, y) and the transformed ones (x', y') is given by:

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

This results in the following equations:

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

2.17.8.2 📋 Task (VPL specification)

Input:

The first line contains `L`.

The second line contains `C`.

The third line contains the factors `shx` and `shy`.

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

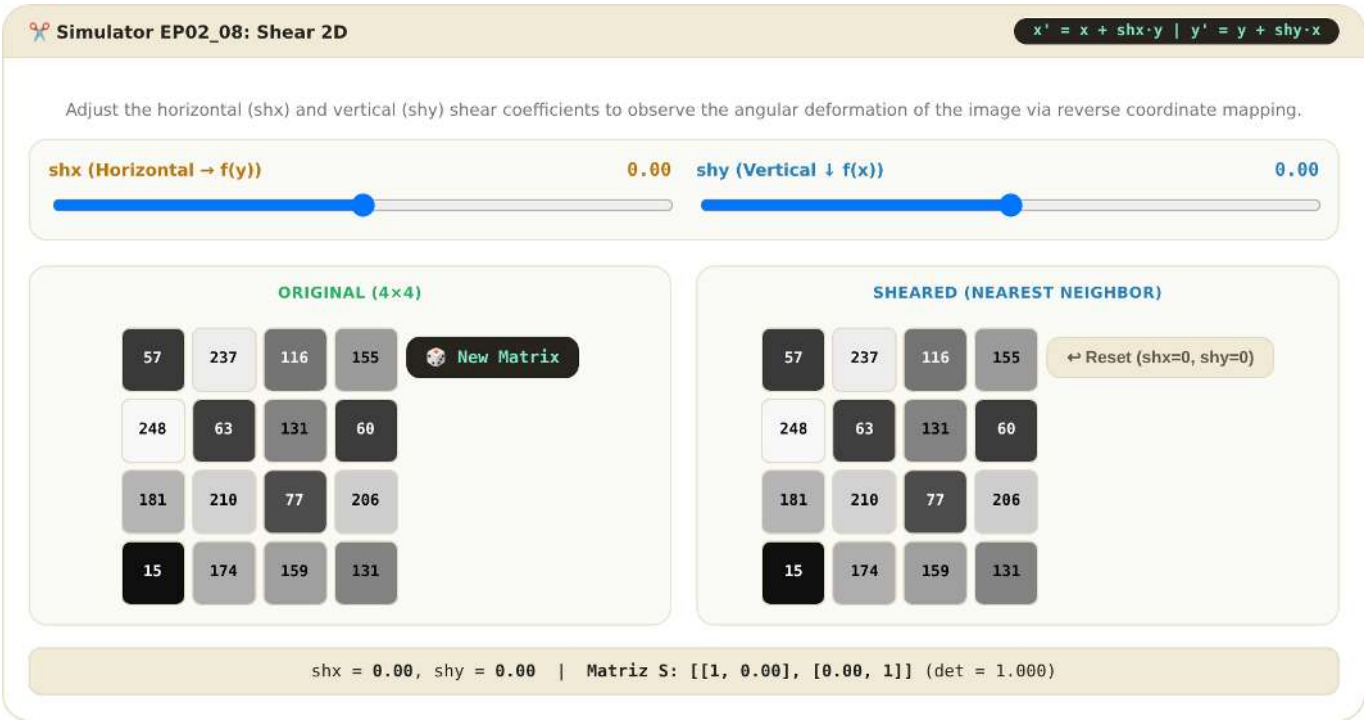
The following lines contain the elements of the $L \times C$ matrix.

Output:

The transformed matrix with the same dimensions $L \times C$.

2.17.8.3 📌 Examples

Input	Output	Observation
3 3 0.5 0.0 nearest 10 20 30 40 50 60 70 80 90	10 20 30 0 40 50 0 0 70	Horizontal shear: row i shifts by $\lfloor i \cdot 0.5 \rfloor$ pixels. Row 0 → 0px, row 1 → 0px, row 2 → 1px. Pixels shifted out are discarded, and empty positions are filled with 0.
2 2 0.0 1.0 nearest 10 20 30 40	10 0 30 20	Vertical shear: column j shifts down by $\lfloor j \cdot 1.0 \rfloor$ pixels. Column 0 → 0px (unchanged), column 1 → 1px: 20 moves down to $(1, 1)$ and $(0, 1)$ becomes 0.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap02/sim-ep0208-cisalhamento.html

Figure 2.19: EP02_08 Simulator: 2D Shear Geometric Transformation (Horizontal and Vertical Shear)

```
%%writefile EP02_08.cpp
// your solution
```

```
Overwriting EP02_08.cpp
```

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

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

2.17.9 EP02_09 🧩 Generic Affine Transformation

In this activity, you must implement an arbitrary affine transformation on an image. This operation is the generalization of all linear transformations (scaling, rotation, shearing) combined with translation, allowing complex geometric manipulations through a single matrix.

- Read two integers **L** and **C**, representing the dimensions of the matrix.
- Read **six real values** (a, b, t_x, c, d, t_y) that compose the 2×3 affine transformation matrix.
- Read a *string* representing the **interpolation** method (`nearest` or `bilinear`).
- Read the integer values of the original matrix.
- Apply the transformation while maintaining the original size $L \times C$.
- Print the resulting matrix.
- See Figure 2.20 for a simulation of this EP.

📌 **Important**:

- **Inverse Mapping:** To compute the value of each pixel in the destination image, you must use the **inverse** of the provided affine transformation matrix to find the corresponding coordinate in the original image.
- **Filling:** Computed coordinates that fall outside the bounds $[0, L-1]$ and $[0, C-1]$ of the original image must result in a pixel with value **0**.
- **Flexibility:** This implementation must be able to perform any of the previous tasks (translation, rotation, etc.) simply by changing the matrix parameters.

Hint:

```
flags = cv2.INTER_NEAREST if interp == 'nearest' else \
        cv2.INTER_CUBIC   if interp == 'bicubic'  else \
        cv2.INTER_LANCZOS4 if interp == 'lanczos' else \
        cv2.INTER_LINEAR

r = cv2.warpAffine(img, M, (C, L), flags=flags)
```

2.17.9.1 🧠 Combining Operations

The affine transformation preserves points, lines, and planes. In image processing, it maps the position (x, y) to (x', y') following the system:

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

Or, compactly in homogeneous coordinates:

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

2.17.9.2 📋 Task (specification for VPL)

Input:

The first line contains `L`.

The second line contains `C`.

The third line contains six floats: `a b tx c d ty`.

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

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

Output:

The transformed matrix with the original dimensions $L \times C$.

2.17.9.3 📌 Examples

Input	Output	Observation
2 2 1.0 0.0 0.5 0.0 1.0 0.5 bilinear 10 20 30 40	15 20 25 30	Fractional translation ($t_x = 0.5, t_y = 0.5$): each output pixel (i, j) samples the position $(i + 0.5,\ j + 0.5)$ from the input via bilinear interpolation. E.g., $(0, 0)$ interpolates the four neighbors $\rightarrow$ 15.
3 3 2.0 0.0 0.0 0.0 2.0 0.0 nearest 1 2 3 4 5 6 7 8 9	1 1 2 1 1 2 4 4 5	2× scaling via the affine matrix ($a = 2, d = 2$): each output pixel (i, j) samples the position $(2i, 2j)$ from the input with nearest. E.g., $(0, 2) \rightarrow (0, 4)$ outside the image $\rightarrow$ nearest clips to $(0, 2) = 3$… awaiting confirmation of the border logic.

```
%%writefile EP02_09.cpp
// your solution
```

```
Overwriting EP02_09.cpp
```

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

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

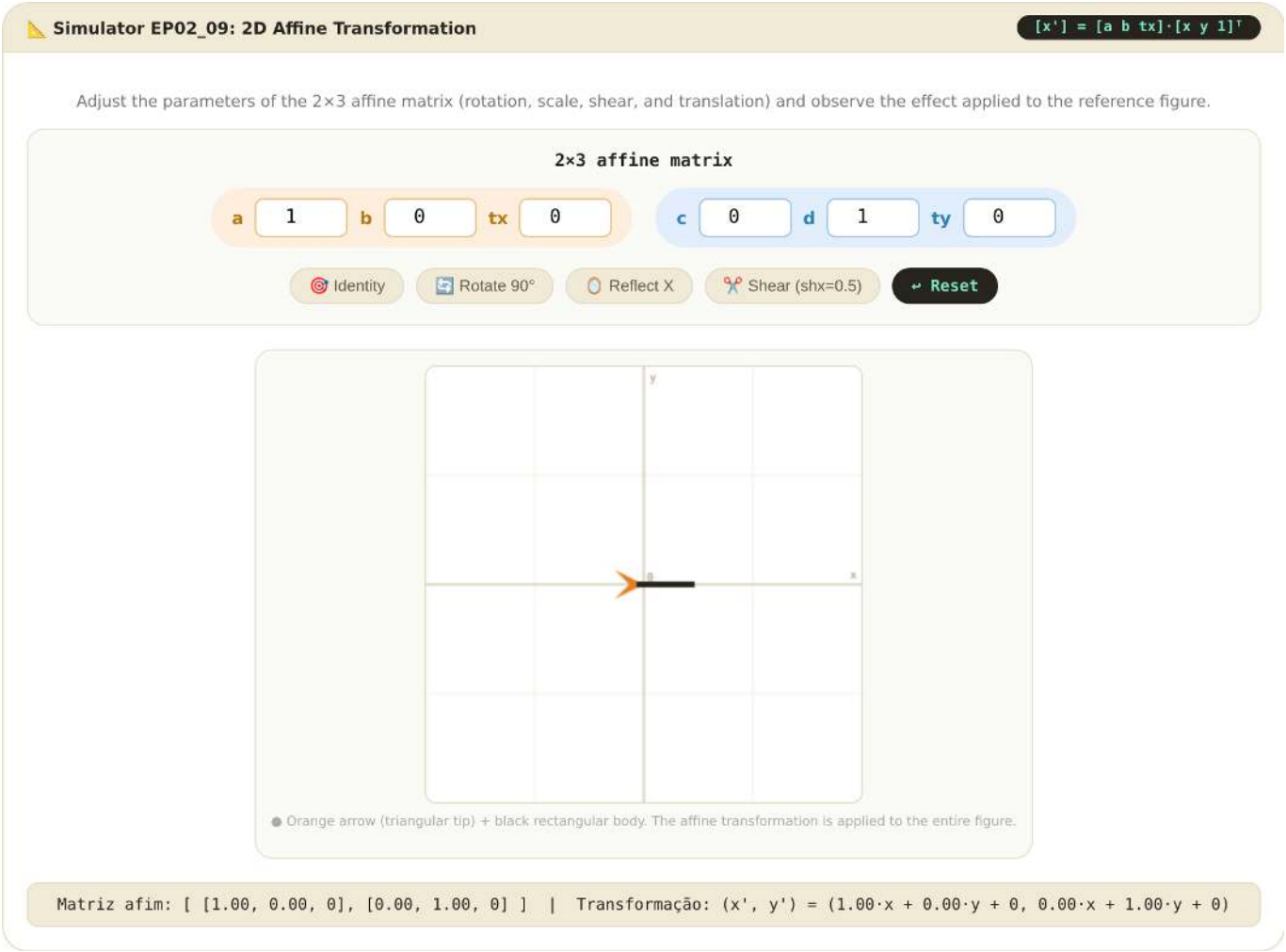
2.17.10 EP02_10 🎯 Perspective Correction (Homography)

In this activity, you must implement the perspective transformation, also known as homography. Unlike affine transformations, perspective does not preserve parallelism, allowing you to "rectify" tilted objects, such as documents or signs captured at oblique angles.

- Read two integers **L** and **C**, representing the dimensions of the original matrix.
- Read **four coordinate pairs** (x, y) representing the corners of the source quadrilateral (distorted object).
- Read **four coordinate pairs** (x, y) representing the corners of the destination quadrilateral (where the object should be mapped).
- Read the values of the original matrix.
- Compute the 3×3 homography matrix and apply the transformation.
- Print the resulting matrix with the specified output dimensions.
- See Figure 2.21 for a simulation of this EP.

📌 **Important**:

- **Degrees of Freedom:** The homography has 8 degrees of freedom (the ninth element of the 3×3 matrix is a normalization constant, usually 1), requiring at least 4 corresponding points to be computed.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap02/sim-ep0209-afim.html

Figure 2.20: Simulator EP02_09: Affine Transformation 2D (2×3 Matrix)

- **Projection:** After multiplying the coordinates by the matrix, you must divide the results x' and y' by the homogeneous component w to return to the 2D plane.
- **Use of Libraries:** For this task, you may use the functions `cv2.getPerspectiveTransform` to obtain the matrix and `cv2.warpPerspective` to apply the transformation, or implement the linear system and inverse mapping manually for an extra challenge.

```
# Output dimensions: bounding box of destination points + 1
w = int(max(pts2[:, 0])) + 1; h = int(max(pts2[:, 1])) + 1
# M = cv2.getPerspectiveTransform(pts1, pts2)
# dst = cv2.warpPerspective(img, M, (w, h))
# or
dst = mm.perspective_transform(img, pts1, pts2, size=(w, h))
```

2.17.10.1 🧠 Non-affine Deformation

While affine transformations map parallelograms to parallelograms, the homography maps any quadrilateral to another quadrilateral. This is essential for computer vision:

Operation	Characteristic	Typical Application
Homography	Plane projection	Document correction, plate scanning.
Vanishing Point	Line convergence	3D reconstruction from 2D images.
Warping	Mesh deformation	Video stabilization and panoramas (stitching).

2.17.10.2 📌 Examples

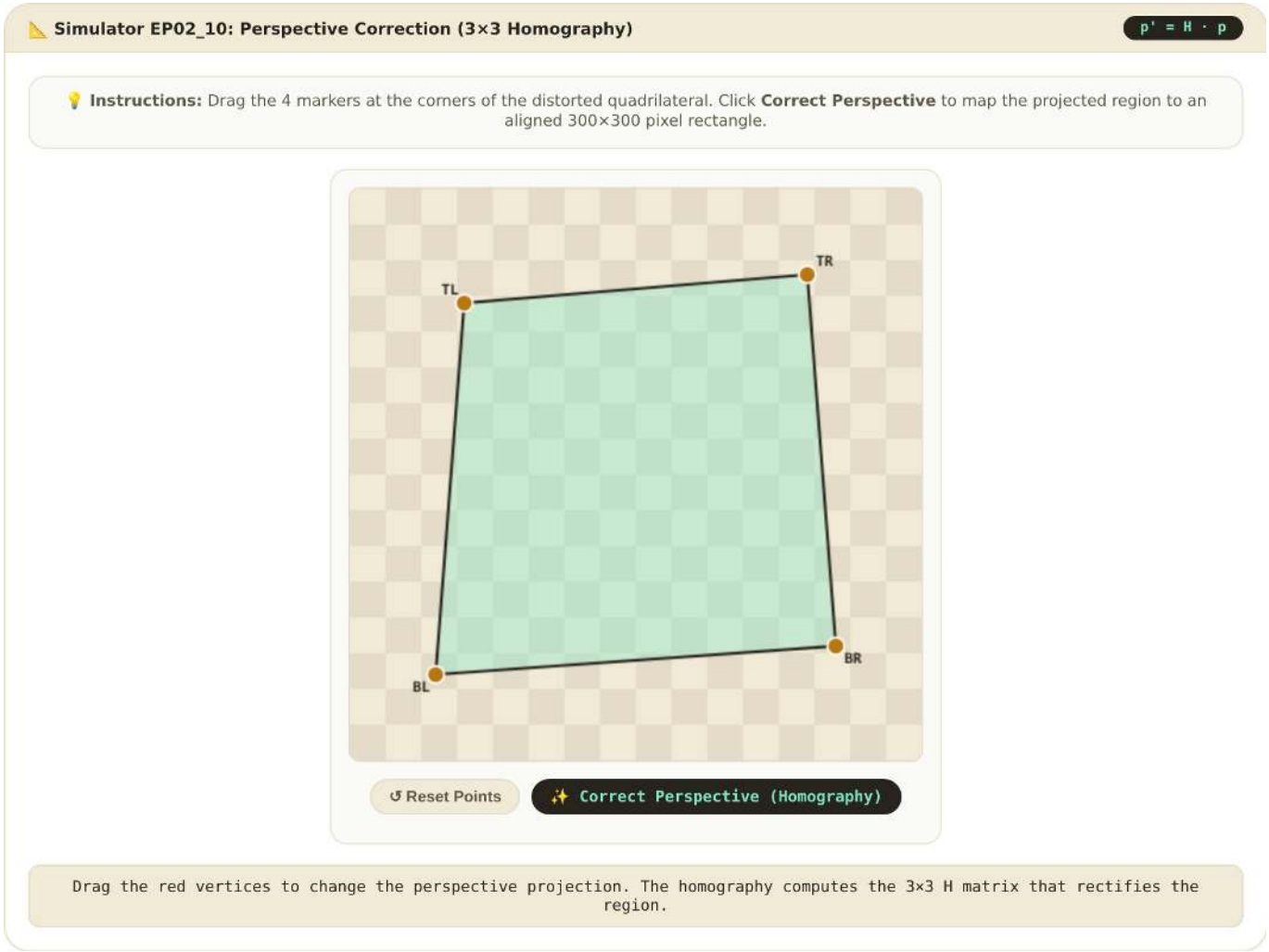
Input	Output	Observation
4 4 0 0 3 0 0 3 3 3 0 0 3 0 0 3 3 3 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160	10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160	The first 4 lines after the dimensions are the source points; the following 4 are the destinations. With identical points, the perspective transformation is the identity and the image is preserved.

```
%%writefile EP02_10.cpp
// your solution
```

```
Overwriting EP02_10.cpp
```

```
TestSuite("EP02_10.cpp").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap02/sim-ep0210-perspectiva.html

Figure 2.21: Simulator EP02_10: Perspective Correction (3×3 Homography Transformation)

2.17.11 EP02_11 🏆 Perspective Correction (Homography) in a Real Image

In this activity, the goal is to apply perspective transformation (homography) to "rectify" a tilted object in a real photograph. You will work with an image of a newspaper, where the grid of a Sudoku puzzle is distorted due to the angle at which the photo was taken.

Your program must read input parameters from the terminal, load the image, compute the 3×3 homography matrix, apply the geometric transformation, and display a global validation indicator.

- Read two integers **L** and **C**, representing the row and column dimensions (height and width) that the rectified output image should have.
- Read **four coordinate pairs** (x, y) from the terminal, representing the four corners of the source quadrilateral (the distorted Sudoku in the original image).
- Automatically compute the **four destination coordinate pairs** using the provided dimensions L and C, mapping the corners to the edges of the new image: $(0, 0)$, $(C-1, 0)$, $(0, L-1)$, and $(C-1, L-1)$.
- Load the local image `sudoku.png` and convert it to grayscale.
- Compute the homography matrix and apply the spatial transformation to the image.
- **Output:** Calculate and print the **sum of all pixels** of the resulting image.

📌 **Important**:

- **Input file:** The image `sudoku.png` must be in the same folder as the script. The program must read it directly from disk (e.g., using `mm::read("sudoku.png")` or `cv2.imread`).
- **Point Order:** Ensure that the reading of the 4 source points and the generation of the 4 destination points strictly follow the same corner order: Top-Left (TL), Top-Right (TR), Bottom-Left (BL), and Bottom-Right (BR).
- **Dimensions in OpenCV:** Remember that functions such as `cv2.warpPerspective` expect the output image size in the format `(width, height)`, which is equivalent to `(C, L)`.
- **Interpolation:** To ensure mathematical consistency of the pixel sum with the automatic grader, use the default bilinear interpolation (`flags=cv2.INTER_LINEAR`).
- **Credits:** The image used is **"Sudoku en periódico"** by Héctor Rodríguez, licensed under **CC BY 2.0**.

2.17.11.1 🧠 Problem Context

The homography has 8 degrees of freedom, requiring at least 4 point correspondences to be computed. Unlike affine transformations, it maps any quadrilateral to another quadrilateral, allowing lines that converge to vanishing points to become parallel again:

Operation	Characteristic	Typical Application
Homography	Projection between planes	Document rectification, scanning of plates and QR Codes.
Inverse Mapping	Scanning from destination to source	Avoids "holes" or empty pixels in the final rectified image.
Warping	Spatial resampling	Correction of lens distortion and panorama stitching.

2.17.11.2 📌 Examples

Input	Output	Observation
500 500 100 120 420 95 80 440 450 460	32982820	The first two inputs are the output dimensions (L and C). The following 4 lines are the (x, y) coordinates of the Sudoku corners in the original image + PAD. The output is the total sum of pixels of the rectified image.
200 200 100 120 420 95 80 440 450 460	5277150	Same source points as the previous example, but generating a smaller output image (200×200). The pixel sum decreases proportionally due to the scale.

2.17.11.3 Sudoku image acquisition and conversion to grayscale

Figure 2.22 shows the reading of the original image followed by its conversion to grayscale and resizing to a 500×500 pixel matrix, preparing the data for the next step.

The perspective correction, applied at Figure 2.23 via the homography matrix, eliminates distortions caused by the camera angle and produces a frontal, regular view of the Sudoku grid.

```
%%writefile tmp/fig_02_sudoku_original.cpp
#define MM_OUT "tmp/fig_02_sudoku_original.png"
//| label: fig-02-sudoku-original
//| fig-cap: "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)."
//| echo: false
//| output: true

#include "morph.hpp"
#include <string>
#include <vector>
#include <filesystem>

int main() {
    std::string url =
"https://upload.wikimedia.org/wikipedia/commons/e/e7/Sudoku_en_peri%C3%B3dico.jpg";
    mm::Image sudoku_rgb = mm::read(url);
    mm::Image sudoku_gray = mm::gray(sudoku_rgb);
    mm::Image sudoku_small = mm::resize(sudoku_gray, 500, 500);
    mm::write(sudoku_small, "sudoku.png");   // consumida pela célula fig-02-sudoku2

    mm::show(
        std::vector<mm::Image>{sudoku_rgb, sudoku_small},
        MM_OUT,
        std::vector<std::string>{"Original RGB", "Cinza 500x500"},
        2
    );

// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(sudoku_rgb, "tmp/fig_02_sudoku_original_0.png");
mm::write(sudoku_small, "tmp/fig_02_sudoku_original_1.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_02_sudoku_original.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_02_sudoku_original.cpp -o tmp/fig_02_sudoku_original \
  && ./tmp/fig_02_sudoku_original \
  && test -f "tmp/fig_02_sudoku_original.png" \
  || echo "  mm::show não gravou tmp/fig_02_sudoku_original.png"
```

```
[1] Original RGB
[2] Cinza 500x500
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_02_sudoku_original_0.png"),
            mm.read("tmp/fig_02_sudoku_original_1.png"),
        ],
        titles=[
            'Original RGB',
            'Cinza 500x500',
        ],
        cols=2,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_02_sudoku_original_0.png (ver a versao Python)")
```

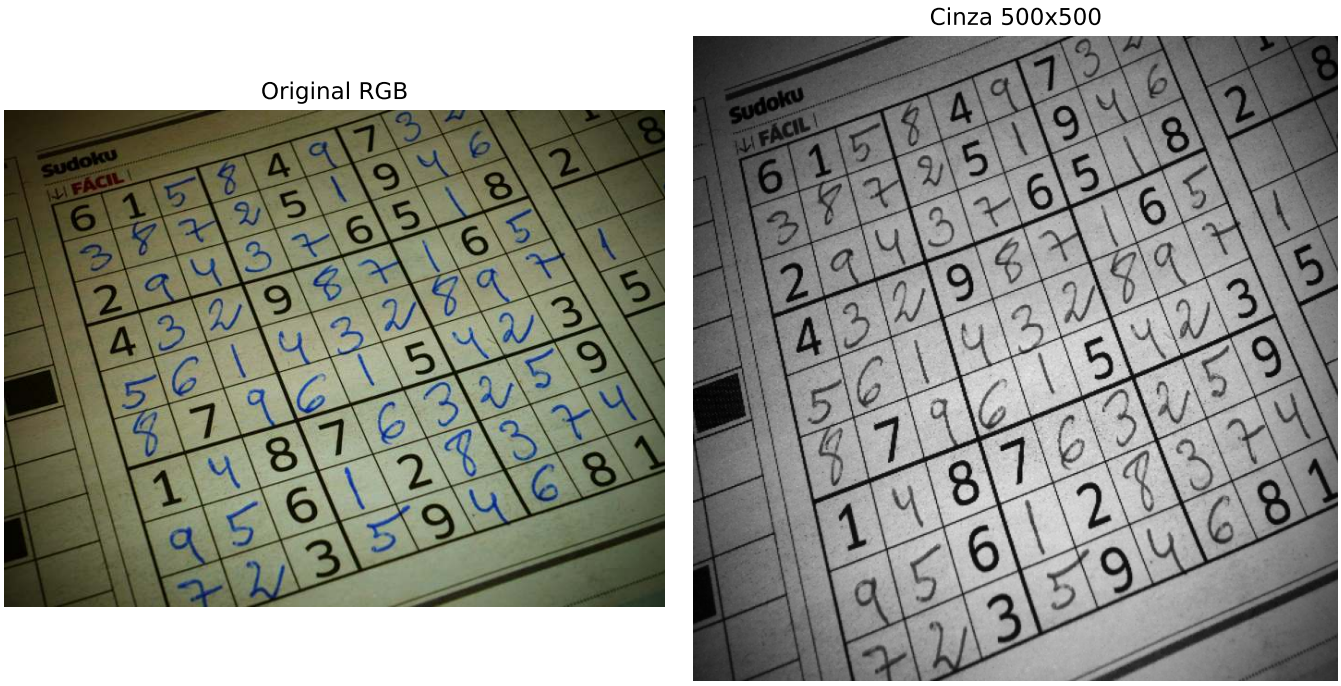


Figure 2.22: Aquisição da imagem de um Sudoku à esquerda. À direita, conversão para tons de cinza e redimensionamento. Crédito: Héctor Rodríguez de Guardamar, Espanha (CC BY 2.0).

```
%%writefile tmp/fig_02_sudoku2.cpp
#define MM_OUT "tmp/fig_02_sudoku2.png"
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <array>
```

```cpp
#include <string>
#include <filesystem>

int main() {
    // 1. Image saved by the previous cell (sudoku.png, 500×500, gray)
    mm::Image img = mm::read("sudoku.png");

    // 2. Padding so the grid corners aren't cut off (mm::pad fills with 0)
    const int PAD = 60;
    mm::Image img_pad = mm::pad(img, PAD);

    // 3. Grid corners in the expanded image - TL, TR, BL, BR
    std::vector<std::array<double, 2>> pts1 = {{100, 160}, {390, 45}, {200, 580}, {570, 420}};

    // 4. Destination corners: front view SIZE×SIZE
    const int SIZE = 500;
    std::vector<std::array<double, 2>> pts2 = {{0, 0}, {SIZE, 0}, {0, SIZE}, {SIZE, SIZE}};

    // 5. Homography pts1 -> pts2 and rectification
    mm::Image img_rect = mm::perspective_transform(img_pad, pts1, pts2, SIZE, SIZE);

    // 6. Display
    mm::show(
        std::vector<mm::Image>{img_pad, img_rect},
        MM_OUT,
        std::vector<std::string>{"Original (with padding)", "Rectified front view"},
        2
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_pad, "tmp/fig_02_sudoku2_0.png");
mm::write(img_rect, "tmp/fig_02_sudoku2_1.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_02_sudoku2.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_02_sudoku2.cpp -o tmp/fig_02_sudoku2 \
  && ./tmp/fig_02_sudoku2 \
  && test -f "tmp/fig_02_sudoku2.png" \
  || echo "  mm::show não gravou tmp/fig_02_sudoku2.png"
```

```
[1] Original (with padding)
[2] Rectified front view
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_02_sudoku2_0.png"),
            mm.read("tmp/fig_02_sudoku2_1.png"),
        ],
        titles=[
            'Original (com padding)',
            'Vista frontal retificada',
        ],
        cols=2,
    )
```

```
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_02_sudoku2_0.png
    (ver a versao Python)")
```

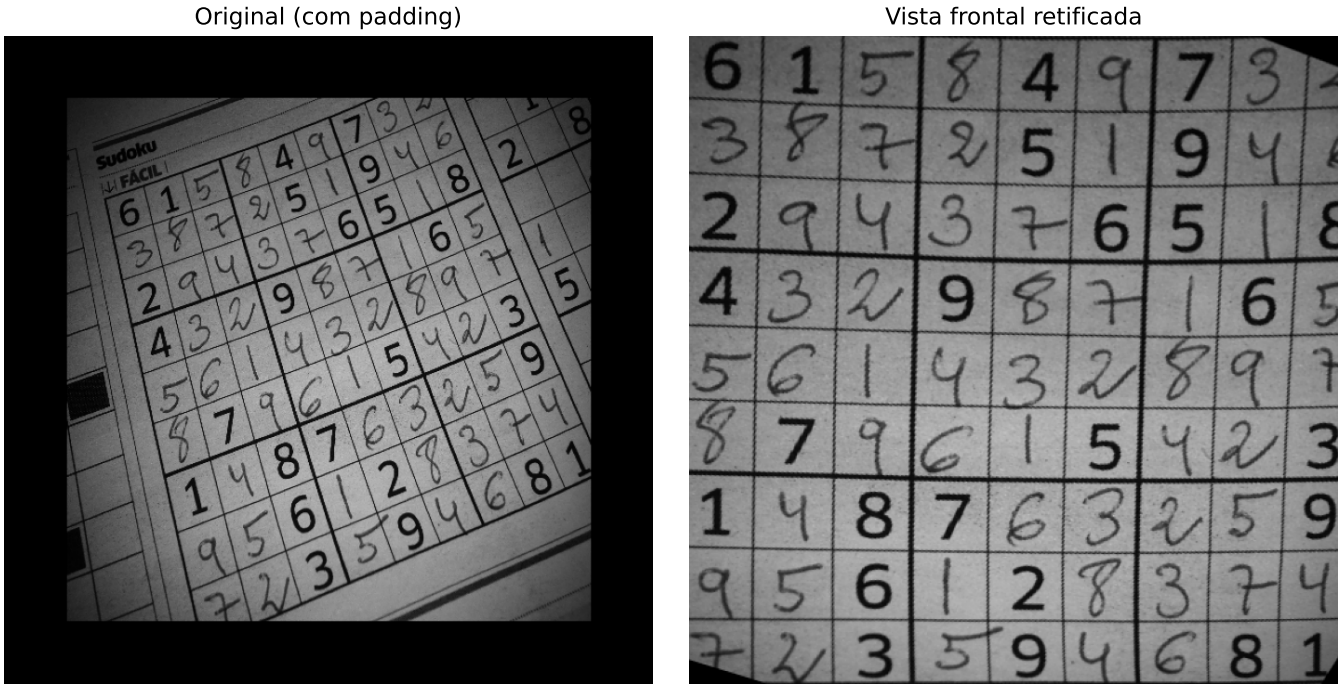


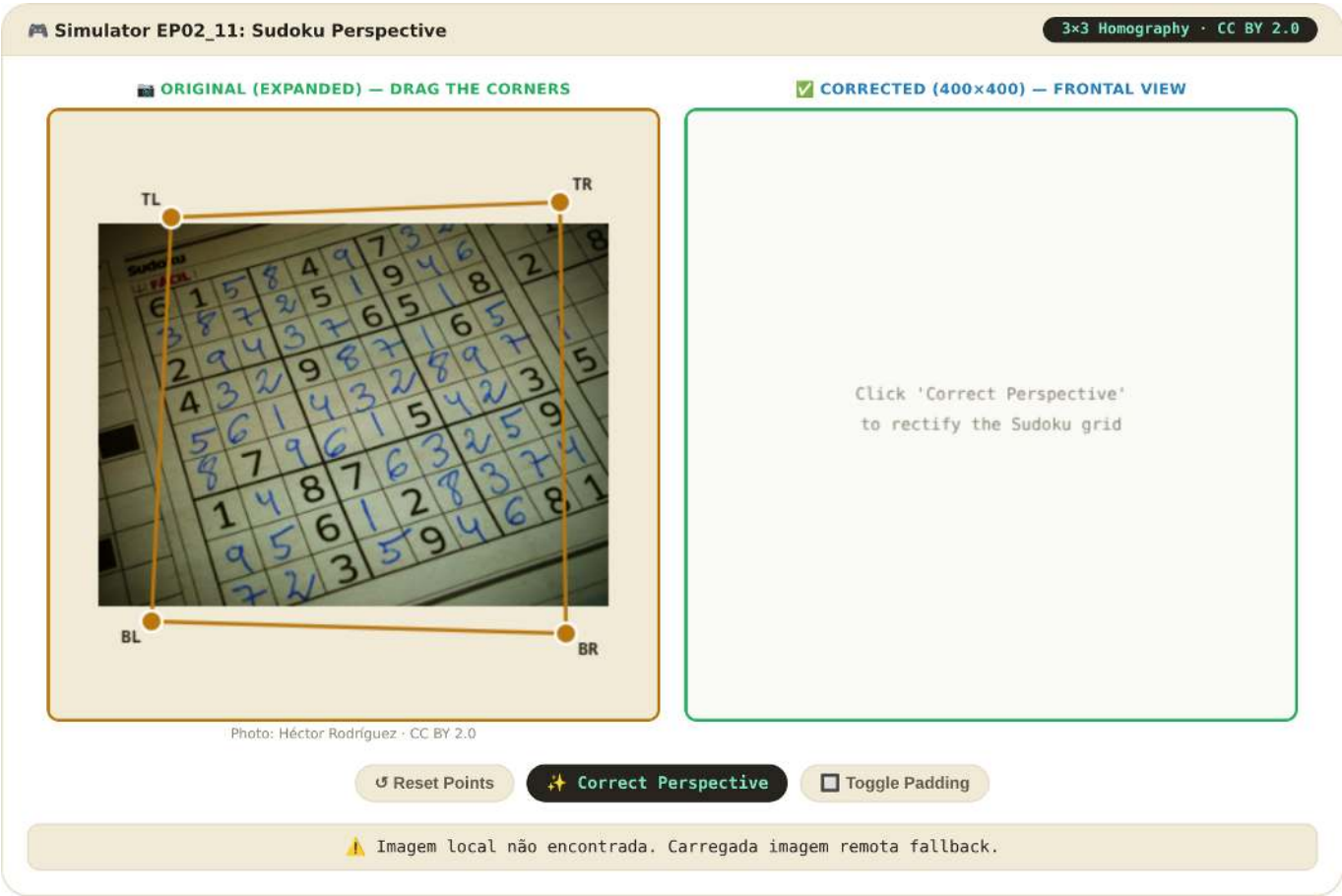
Figure 2.23: Correção de perspectiva por homografia 3×3: imagem com *padding* e a vista frontal retificada, via *mm::perspective_transform* (solve 8×8 dos 4 pontos + mapeamento inverso projetivo).

```
%%writefile EP02_11.cpp
// your solution
```

```
Overwriting EP02_11.cpp
```

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

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap02/sim-ep0211-sudoku.html

Figure 2.24: EP02_11 Simulator: Sudoku Perspective Correction (3×3 Homography with Bilinear Resampling)

Chapter 3

Spatial Operations: Intensity, Histogram, and Filtering

This chapter delves into image processing in the spatial domain, starting from the direct manipulation of pixels and histograms for contrast enhancement, to the application of local filters via convolution for smoothing, noise reduction, and edge detection. The goal is to develop the mathematical and computational intuition that underpins much of modern Computer Vision algorithms.

3.1 Objectives

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

- **Manipulate intensity and pixels:** Perform saturated arithmetic operations (`mm::addm`, `mm::subm`) and bitwise logic operations (`mm::band`, `mm::bor`, `mm::bnot`) for combining and selecting regions of interest (ROI), and apply *alpha blending* (`mm::blend`) for weighted image fusion;
- **Process histograms:** Interpret the histogram as a tonal diagnostic and apply global equalization via CDF (`mm::equalize`); in the Python track, also adaptive equalization (CLAHE) and histogram specification for tonal profile transfer between images;
- **Understand spatial fundamentals:** Understand neighborhood, border *padding* (`mm::pad`), and the difference between cross-correlation (`mm::conv`) and convolution — including why asymmetric kernels such as Sobel produce distinct results in the two operations;
- **Apply smoothing filtering:** Use the mean filter (`mm::blur`, or `mm::conv` with a uniform kernel) and the Gaussian filter (`mm::gaussian`) for noise reduction, understanding the advantage of radial weighting and Gaussian separability;
- **Apply enhancement filtering:** Use the Laplacian w_4 and w_8 (`mm::laplacian`) for isotropic edge enhancement, the Sobel operator (`mm::sobel`) for gradient magnitude — and, in the Python track, the directional decomposition G_x, G_y, and angle —, and *Unsharp Masking* (`mm::usm`) for high-frequency amplification controlled by the parameter k;
- **Use order filters:** Apply the median filter (`mm::median`) for salt-and-pepper noise removal, understanding why its nonlinear nature and robustness to *outliers* make it superior to linear filters in this scenario;
- **Solve practical problems:** Chain techniques into preprocessing *pipelines* (equalization → Gaussian → Canny; with CLAHE replacing equalization in the Python track) and use the `morph` functions (`mm::conv`, `mm::histImg`, `mm::equalize`, `mm::drawImgKernel`) for didactic analysis and visualization of each step.

3.2 Intensity Level Operations

The most elementary level of image processing acts directly on pixel values, without considering neighborhood. These operations—called **point operations**—are the fastest computationally and form the basis for more complex techniques.

Formally, a point transformation can be described as:

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

where $f(x, y)$ is the input image, $g(x, y)$ is the output, and T is a function applied to each pixel individually.

3.2.1 Preparing the Practical Environment

The following block loads the `morph` library from the repository (the `morph.py` module and, in the C++ track, also the `morph.hpp` used in the `#include` of compiled cells).

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # C++ track build artifacts (.cpp, binary, PNGs)

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")

# The kernel is Python even on the C++ track: `mm` (morph.py) is used by the
# simulators, by the display of the figures the C++ binary generates, and by
# the mm::Image state between cells. cpp=True also downloads the compiled track
# (morph.hpp + stb_image*.h), used in the #include of the %%writefile *.cpp cells.
import config
config.setup(cpp=True)
from morph import mm
import numpy as np
```

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

As the object of study throughout this chapter, we will use the wildlife images presented in Figure 3.1 and Figure 3.3. From them, we will explore spatial operations on intensity, histograms, and filtering, analyzing their effects on enhancement, smoothing, noise reduction, and edge detection, in order to understand the mathematical and computational fundamentals of DIP.

```
%%writefile tmp/fig_03_mandrill.cpp
#define MM_OUT "tmp/fig_03_mandrill.png"
//| label: fig-03-mandrill
//| fig-cap: "*Mandrill* (*Mandrillus sphinx*) photographed in its natural environment in
South Africa. Credit: Carlos Guilherme Rodrigues (CC BY-SA 3.0)."
//| echo: true

#include "morph.hpp"
#include <iostream>
#include <filesystem>

int main() {
    mm::Image img_color =
mm::read("https://upload.wikimedia.org/wikipedia/commons/9/9b/Carlos_Guilherme_Rodrigues_%2876515283%29.
    mm::Image img_gray = mm::gray(img_color);

    mm::show(img_color, MM_OUT);


// [pdi:state-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp/state");
mm::write(img_gray, "tmp/state/img_gray_8.png");
// [pdi:state-io:end]
return 0;
```

```
}
```

```
Overwriting tmp/fig_03_mandrill.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_mandrill.cpp -o tmp/fig_03_mandrill \
  && ./tmp/fig_03_mandrill \
  && test -f "tmp/fig_03_mandrill.png" \
  || echo "  mm::show não gravou tmp/fig_03_mandrill.png"
```

```
try:
    mm.show(mm.read("tmp/fig_03_mandrill.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_mandrill.png
    (ver a versao Python)")
```

Figure 3.1: *Mandrill* (*Mandrillus sphinx*) fotografado em ambiente natural na África do Sul. Crédito: Carlos Guilherme Rodrigues (CC BY-SA 3.0).

3.2.2 Arithmetic Operations

Arithmetic operations between images are widely used in DIP to combine, compare, or enhance information. **Image subtraction** is especially powerful for detecting differences between two frames — for example, in removing static backgrounds in surveillance cameras:

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

Saturated addition limits the result to the interval $[0, 255]$: values above 255 are clamped to 255, avoiding the silent *overflow* of the `uint8` type (e.g., $200 + 100 = 44$ instead of 300). **Saturated subtraction** applies the same principle on the lower side: negative values are clamped to 0.

> ⚠ Saturation and *overflow*
>
> Arithmetic operations on `uint8` suffer from silent *overflow*: $200 + 100 = 44$ (not 300). `mm::addm` and `mm::subm` perform **automatic saturation**, clamping the result to $[0, 255]$. *Blending* uses fractional weights: `mm::blend` operates internally in floating point and only then rounds and saturates to `uint8`.

Figure 3.2 demonstrates the addition of a constant (brightening) and the subtraction of a constant (darkening with saturation at 0).

```
%%writefile tmp/fig_03_aritmetica.cpp
#define MM_OUT "tmp/fig_03_aritmetica.png"
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <filesystem>
```

```
int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_8.png");
// [pdi:state-io:end]

    //
    // Operations: log transformation, gamma correction, and addition.
    //

    int fundo = 60;

    mm::Image img_add = mm::addm(img_gray, fundo);
    mm::Image img_sub = mm::subm(img_gray, fundo);

    mm::show(
        std::vector<mm::Image>{img_gray, img_add, img_sub},
        MM_OUT,
        std::vector<std::string>{"Original", "addm (+60)", "subm (-60)"},
        3
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray, "tmp/fig_03_aritmetica_0.png");
mm::write(img_add, "tmp/fig_03_aritmetica_1.png");
mm::write(img_sub, "tmp/fig_03_aritmetica_2.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_03_aritmetica.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_aritmetica.cpp -o tmp/fig_03_aritmetica \
  && ./tmp/fig_03_aritmetica \
  && test -f "tmp/fig_03_aritmetica.png" \
  || echo "  mm::show não gravou tmp/fig_03_aritmetica.png"
```

```
[1] Original
[2] addm (+60)
[3] subm (-60)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_03_aritmetica_0.png"),
            mm.read("tmp/fig_03_aritmetica_1.png"),
            mm.read("tmp/fig_03_aritmetica_2.png"),
        ],
        titles=[
            'Original',
            'addm (+60)',
            'subm (-60)',
        ],
        cols=3,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_03_aritmetica_0.png (ver a versao Python)")
```

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

3.2.3 Weighted Mixture (*Alpha Blending*)

The **weighted mixture** (*alpha blending*) combines two images using complementary weights α and $(1-\alpha)$:

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

When $\alpha = 1$, only image f_1 is obtained; when $\alpha = 0$, only f_2. Intermediate values produce a smooth transition between the two, being widely used in image composition, layer overlaying, watermarks, and visual blending effects.

For the combination to produce a coherent result, it is necessary to align the regions of interest beforehand. In Figure 3.4, the leopard's face is cropped with `mm::crop(img_leop_gray, 250, H-300, 100, W-200)` and the mandrill's facial region with `mm::crop(img_gray, 100, 400, 380, 530)`, so that the eyes and facial structure are approximately aligned. The leopard crop is then resized (`mm::resize`) to the mandrill's dimensions before the blending.

`mm::blend` performs the operation in floating point — avoiding *overflow* in the calculations with fractional weights — and only then rounds and saturates the result to `uint8`.

```
%%writefile tmp/fig_03_leopardo.cpp
#define MM_OUT "tmp/fig_03_leopardo.png"
//| label: fig-03-leopardo
//| fig-cap: "Retrato de um leopardo (*Panthera pardus*) em ambiente natural. Crédito: C.
Brück (CC BY-SA 4.0)."
//| echo: true

#include "morph.hpp"
#include <filesystem>

int main() {
    mm::Image img_leop =
mm::read("https://upload.wikimedia.org/wikipedia/commons/9/92/Leopard_%28Panthera_pardus%29_portrait.jpg
    mm::Image img_leop_gray = mm::gray(img_leop);

    mm::show(img_leop, MM_OUT);


// [pdi:state-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp/state");
mm::write(img_leop, "tmp/state/img_leop_12.png");
mm::write(img_leop_gray, "tmp/state/img_leop_gray_12.png");
// [pdi:state-io:end]
return 0;
}
```

```
Overwriting tmp/fig_03_leopardo.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_leopardo.cpp -o tmp/fig_03_leopardo \
  && ./tmp/fig_03_leopardo \
  && test -f "tmp/fig_03_leopardo.png" \
  || echo "  mm::show não gravou tmp/fig_03_leopardo.png"
```

```
try:
    mm.show(mm.read("tmp/fig_03_leopardo.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_leopardo.png
    (ver a versao Python)")
```

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

```
%%writefile tmp/fig_03_blend.cpp
#define MM_OUT "tmp/fig_03_blend.png"
//| label: fig-03-blend
//| fig-cap: "*Alpha blending* entre recortes alinhados de mandrill e do leopardo
(@fig-03-leopardo) para diferentes valores de  . Em  =1 vê-se apenas mandrill; em  =0, apenas o
leopardo; valores intermediários fundem os olhares das duas imagens proporcionalmente."
//| echo: true
//| output: true

#include "morph.hpp"
#include <vector>
#include <string>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_8.png");
mm::Image img_leop_gray = mm::_read_state("tmp/state/img_leop_gray_12.png");
// [pdi:state-io:end]

    // Recortes alinhados: rosto do leopardo e região facial do mandril
```

```
    mm::Image leo = mm::crop(img_leop_gray, 250, img_leop_gray.h - 300, 100, img_leop_gray.w -
200);
    mm::Image mandrill = mm::crop(img_gray, 100, 400, 380, 530);
    mm::Image leo_r = mm::resize(leo, mandrill.w, mandrill.h, "bilinear");

    mm::show(
        std::vector<mm::Image>{mm::blend(mandrill, leo_r, 1.0), mm::blend(mandrill, leo_r,
0.8),
                               mm::blend(mandrill, leo_r, 0.6), mm::blend(mandrill, leo_r,
0.4),
                               mm::blend(mandrill, leo_r, 0.2), mm::blend(mandrill, leo_r,
0.0)},
        MM_OUT,
        std::vector<std::string>{" =1.0", " =0.8", " =0.6", " =0.4", " =0.2", " =0.0"},
        6
    );

    return 0;
}
```

```
Overwriting tmp/fig_03_blend.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_blend.cpp -o tmp/fig_03_blend \
  && ./tmp/fig_03_blend \
  && test -f "tmp/fig_03_blend.png" \
  || echo "  mm::show não gravou tmp/fig_03_blend.png"
```

```
[1]  =1.0
[2]  =0.8
[3]  =0.6
[4]  =0.4
[5]  =0.2
[6]  =0.0
```

```
try:
    mm.show(mm.read("tmp/fig_03_blend.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_blend.png (ver
    a versao Python)")
```

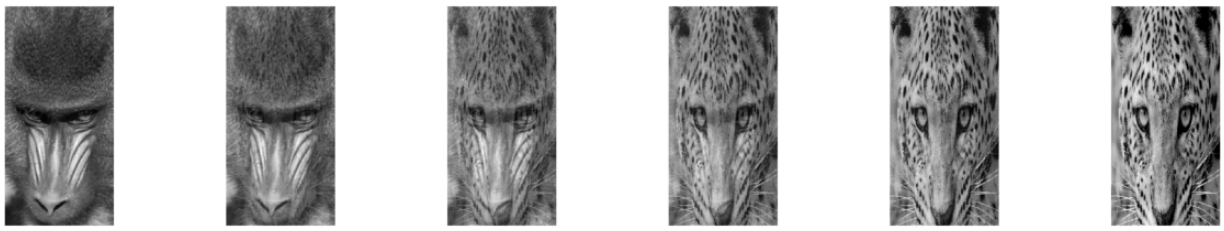


Figure 3.4: *Alpha blending* entre recortes alinhados de mandrill e do leopardo (Figure 3.3) para diferentes valores de . Em =1 vê-se apenas mandrill; em =0, apenas o leopardo; valores intermediários fundem os olhares das duas imagens proporcionalmente.

3.2.4 Logical Operations and Bitwise Masks

Bitwise logical operations (AND, OR, and NOT) act directly on the bits of each pixel and are the foundation for creating and applying **masks** (*masks*) — binary images with only 0 (black) and 255 (white) used to isolate **Regions of Interest** (**ROI**).

The behavior of each operation stems from the binary representation of 255 (`11111111`) and 0 (`00000000`):

- **AND** with the mask: where $m = 255$, the original bits are preserved; where $m = 0$, the pixel is zeroed. Result: ROI cropping.

$$g(x, y) = f(x, y) \text{ AND } m(x, y) \tag{3.4}$$

- **OR** with the mask: where $m = 255$, the pixel is forced to white; where $m = 0$, the original value is retained. Result: ROI highlighting.
- **NOT** (without mask): inverts all bits ($g = 255 - f$), producing the photographic negative of the image.

Figure 3.5 illustrates the three operations applied to the mandrill image with a circular mask.

```
%%writefile tmp/fig_03_logica.cpp
#define MM_OUT "tmp/fig_03_logica.png"
#include "morph.hpp"
#include <iostream>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_8.png");
// [pdi:state-io:end]

    //| label: fig-03-logica
    //| fig-cap: "Binary bitwise operations with circular mask: AND (ROI isolation), OR (ROI
    illumination) and NOT (negative)."
    //| echo: true
    //| output: true

    int h = img_gray.h;
    int w = img_gray.w;

    // Filled circular mask, centered on the image (mm.circle draws the
    // disk; the didactic version, pixel-by-pixel radius test, is mm.circle0).
    mm::Image mask_circ(h, w);
    mask_circ = mm::circle(mask_circ, w / 2, h / 2, std::min(h, w) / 3 - 10, 255, -1);

    // Operations via morph
    mm::Image img_not = mm::bnot(img_gray);            // NOT: photographic negative
    mm::Image img_and = mm::band(img_gray, mask_circ); // preserves only the circular ROI
    mm::Image img_or  = mm::bor(img_gray, mask_circ);  // illuminates the mask region

    mm::show(
        std::vector<mm::Image>{img_gray, img_and, img_or, img_not},
        MM_OUT,
        std::vector<std::string>{"Original", "AND (circular ROI)", "OR (illuminates ROI)",
"NOT (negative)"},
        4
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray, "tmp/fig_03_logica_0.png");
mm::write(img_and, "tmp/fig_03_logica_1.png");
mm::write(img_or, "tmp/fig_03_logica_2.png");
mm::write(img_not, "tmp/fig_03_logica_3.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_03_logica.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_logica.cpp -o tmp/fig_03_logica \
  && ./tmp/fig_03_logica \
  && test -f "tmp/fig_03_logica.png" \
  || echo "  mm::show não gravou tmp/fig_03_logica.png"
```

```
[1] Original
[2] AND (circular ROI)
[3] OR (illuminates ROI)
[4] NOT (negative)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_03_logica_0.png"),
            mm.read("tmp/fig_03_logica_1.png"),
            mm.read("tmp/fig_03_logica_2.png"),
            mm.read("tmp/fig_03_logica_3.png"),
        ],
        titles=[
            'Original',
            'AND (ROI circular)',
            'OR (ilumina ROI)',
            'NOT (negativo)',
        ],
        cols=4,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_logica_0.png
    (ver a versao Python)")
```

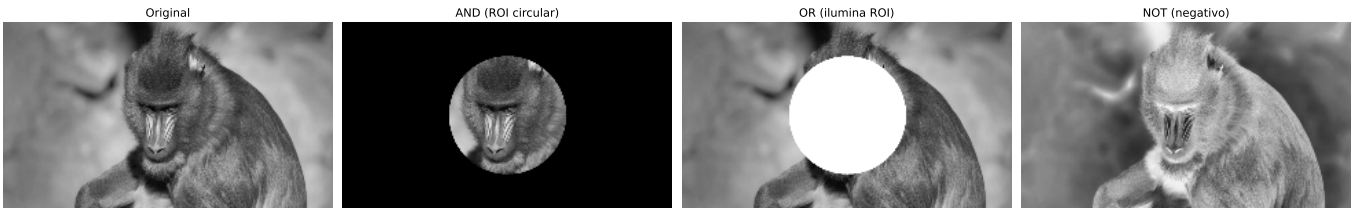


Figure 3.5: Operações lógicas bit a bit com máscara circular: AND (isolamento da ROI), OR (iluminação da ROI) e NOT (negativo).

3.3 Image Histogram

The **histogram** of a grayscale image is a discrete function that describes the frequency distribution of intensities:

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

where r_k is the k-th intensity level, n_k is the number of pixels with that intensity, and L is the total number of levels (typically 256 for 8 bits). The normalized histogram estimates the probability of each level:

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

where MN is the total number of pixels. As a **global statistic**, the histogram does not carry positional information, but it reveals essential characteristics such as average brightness, contrast, and tonal distribution. In practice, `mm::hist(img)` returns the count vector $h(r_k)$, which serves both for visualization (via `mm::histImg`) and for calculations such as the **cumulative distribution function (CDF)** and equalization.

i Histogram Interpretation

- **Narrow on the left:** underexposed image (dark).
- **Narrow on the right:** overexposed image (bright).
- **Concentrated in the center:** low contrast.
- **Spread across the entire range:** high contrast, good utilization of the available tones.

Figure 3.6 presents the histogram of the mandrill image, as well as darkened (`mm::subm`) and brightened (`mm::addm`) versions. The shift of the intensity distribution to the left and to the right is observed, respectively. Note that the range represented on the x-axis does not necessarily correspond to the entire 0 to 255 range.

```
%%writefile tmp/fig_03_histograma.cpp
#define MM_OUT "tmp/fig_03_histograma.png"
//| label: fig-03-histograma
//| fig-cap: "Histogramas da imagem original, de uma versão escurecida (-80) e de uma clareada
(+80). A subtração/adição satura em 0 e 255."
//| echo: true
//| output: true

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_8.png");
// [pdi:state-io:end]

    mm::Image img_dark = mm::subm(img_gray, 80);
    mm::Image img_high = mm::addm(img_gray, 80);

    mm::show(
        std::vector<mm::Image>{img_gray, img_dark, img_high,
                               mm::histImg(img_gray), mm::histImg(img_dark),
mm::histImg(img_high)},
        MM_OUT,
        std::vector<std::string>{"Original", "Escurecida (-80)", "Clareada (+80)",
                                 "Histograma - original", "Histograma - escurecida",
                                 "Histograma - clareada"},
        3
    );

    return 0;
}
```

```
Overwriting tmp/fig_03_histograma.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_histograma.cpp -o tmp/fig_03_histograma \
  && ./tmp/fig_03_histograma \
  && test -f "tmp/fig_03_histograma.png" \
  || echo "  mm::show não gravou tmp/fig_03_histograma.png"
```

```
[1] Original
[2] Escurecida (-80)
[3] Clareada (+80)
[4] Histograma - original
[5] Histograma - escurecida
[6] Histograma - clareada
```

```
try:
    mm.show(mm.read("tmp/fig_03_histograma.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_histograma.png
    (ver a versao Python)")
```

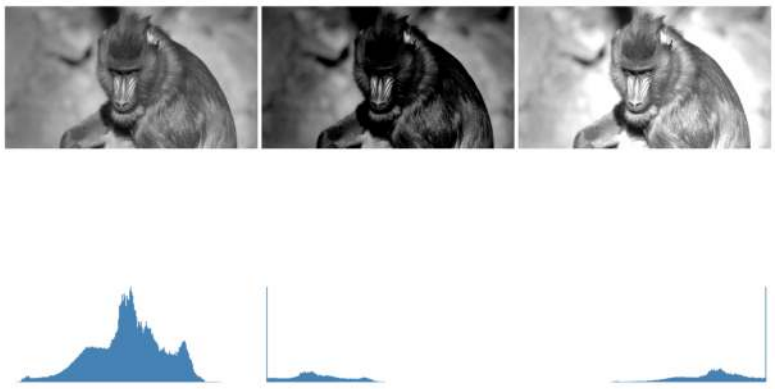


Figure 3.6: Histogramas da imagem original, de uma versão escurecida (-80) e de uma clareada ($+80$). A subtração/adição satura em 0 e 255.

3.3.1 Histogram Equalization

Histogram equalization redistributes intensities so that the resulting histogram is as uniform as possible. The mapping is given by the **cumulative distribution function (CDF)**:

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p(r_j) = \frac{L-1}{MN}\sum_{j=0}^{k} n_j \tag{3.7}$$

The transformation is monotonic: frequent levels receive larger intervals in the output domain (greater separation → more contrast), while rare levels are compressed.

The complete algorithm, in five steps, is presented in Table 3.1.

Table 3.1: Histogram equalization algorithm.

Step	Operation	Formula
1	**Histogram**	$h[k] \leftarrow$ number of pixels with intensity k, $k = 0 \dots L-1$
2	**Probability**	$p[k] \leftarrow h[k]/MN$
3	**CDF**	$\text{cdf}[k] \leftarrow \sum_{j=0}^{k} p[j]$ (cumulative sum)
4	***Look-Up Table* (mapping)**	$\text{lut}[k] \leftarrow \text{round}(\text{cdf}[k] \times (L-1))$

Step	Operation	Formula
5	**Application**	$g[i,j] \leftarrow \text{lut}[f[i,j]]$ (for every pixel)

Note in Figure 3.7 that equalization **redistributes** the existing tones to more spaced positions in the range $[0, L-1]$, but it does not create new tones — the equalized image still has exactly 3 distinct tones, now at $\{1,5,7\}$ instead of $\{2,3,4\}$.

```
%%writefile tmp/fig_03_equalizacao_didatica.cpp
#define MM_OUT "tmp/fig_03_equalizacao_didatica.png"
#include "morph.hpp"
#include <iostream>
#include <vector>

//| label: fig-03-equalizacao-didatica
//| fig-cap: "Equalização de histograma numa imagem 5×5 de 3 bits (L=8): tons concentrados em
{2,3,4} são redistribuídos pela CDF. *mm::equalize(img, 3)* faz o mapeamento."
//| echo: true
//| output: true

int main() {
    mm::Image img5(5, 5);
    int vals[5][5] = {{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}};
    for (int y = 0; y < 5; y++)
        for (int x = 0; x < 5; x++)
            img5.at(y, x) = vals[y][x];

    mm::Image img5_eq = mm::equalize(img5, 3);   // L = 2^3 = 8

    std::cout << "Imagem original 5x5 (3 bits):\n";
    std::cout << mm::drawImg(img5);
    std::cout << "Imagem equalizada 5x5:\n";
    std::cout << mm::drawImg(img5_eq);

    mm::show(std::vector<mm::Image>{img5, img5_eq, mm::histImg(img5), mm::histImg(img5_eq)},
             MM_OUT,
             std::vector<std::string>{"Original", "Equalizada", "Histograma - original",
"Histograma - equalizada"},
             2);

    return 0;
}
```

```
Overwriting tmp/fig_03_equalizacao_didatica.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_equalizacao_didatica.cpp -o tmp/fig_03_equalizacao_didatica \
  && ./tmp/fig_03_equalizacao_didatica \
  && test -f "tmp/fig_03_equalizacao_didatica.png" \
  || echo "  mm::show não gravou tmp/fig_03_equalizacao_didatica.png"
```

```
Imagem original 5x5 (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
Imagem equalizada 5x5:
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
[1] Original
[2] Equalizada
[3] Histograma - original
[4] Histograma - equalizada
```

```
try:
    mm.show(mm.read("tmp/fig_03_equalizacao_didatica.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_03_equalizacao_didatica.png (ver a versao Python)")
```

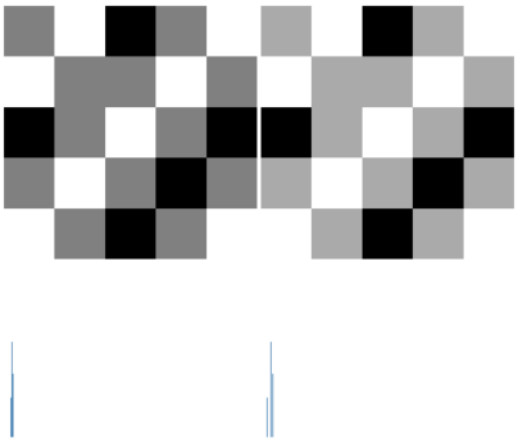


Figure 3.7: Equalização de histograma numa imagem 5×5 de 3 bits (L=8): tons concentrados em {2,3,4} são redistribuídos pela CDF. *mm::equalize(img, 3)* faz o mapeamento.

Limitation: global equalization can over-enhance noise and produce excessive contrast in homogeneous regions. **CLAHE** (*Contrast Limited Adaptive Histogram Equalization*) reduces this problem by applying equalization in local blocks (*tiles*) and limiting the height of histogram peaks before equalization.

Figure 3.8 compares the original image, the global equalization via `mm::equalize`, and OpenCV's CLAHE, also displaying the resulting histograms. Unlike global equalization, which uses a single transformation based on the CDF of the entire image, CLAHE adapts contrast to each region, being particularly useful in images with non-uniform illumination.

In the example, `clipLimit=2.0` and `tileGridSize=(32,32)` were used. The `clipLimit` parameter defines how much the local histogram peaks can grow before being clipped. In OpenCV, this value is a relative factor: the actual limit is roughly calculated as `clipLimit × (number of pixels in the block / number of gray levels)`. For example, in a block with 4096 pixels and an 8-bit image (256 gray levels), the average frequency per level is $4096/256 = 16$. Thus, `clipLimit=2.0` allows peaks of approximately $2 \times 16 = 32$ occurrences before clipping. The excess occurrences are not discarded: they are redistributed among the remaining gray levels of the histogram, reducing excessive concentration in few levels and avoiding exaggerated amplification of local contrast. Smaller values limit contrast more and reduce noise amplification, while larger values allow more intense enhancement but may introduce artifacts.

- `clipLimit=1.0`: smooth and conservative enhancement;
- `clipLimit=2.0`: good balance between contrast and naturalness;
- `clipLimit=4.0`: greater emphasis on local details;
- `clipLimit=8.0`: aggressive contrast, with possible noise amplification.

Thus, CLAHE usually produces more natural results than global equalization, especially in images with shadows, reflections, or uneven illumination.

```
%%writefile tmp/fig_03_equalizacao.cpp
#define MM_OUT "tmp/fig_03_equalizacao.png"
//| label: fig-03-equalizacao
//| fig-cap: "Equalização de histograma global (mm::equalize, via CDF) e os histogramas
antes/depois. CLAHE (adaptativa) fica só na trilha Python - não tem equivalente em morph.hpp."
//| echo: true
//| output: true

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_8.png");
// [pdi:state-io:end]

    // img_gray is assumed to be already initialized

    mm::Image img_eq = mm::equalize(img_gray);

    mm::show(
        std::vector<mm::Image>{img_gray, img_eq, mm::histImg(img_gray), mm::histImg(img_eq)},
        MM_OUT,
        std::vector<std::string>{"Original", "mm.equalize (CDF)", "Histograma - original",
"Histograma - equalizado"},
        2
    );

    return 0;
}
```

```
Overwriting tmp/fig_03_equalizacao.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_equalizacao.cpp -o tmp/fig_03_equalizacao \
  && ./tmp/fig_03_equalizacao \
  && test -f "tmp/fig_03_equalizacao.png" \
  || echo "  mm::show não gravou tmp/fig_03_equalizacao.png"
```

```
[1] Original
[2] mm.equalize (CDF)
[3] Histograma - original
[4] Histograma - equalizado
```

```
try:
    mm.show(mm.read("tmp/fig_03_equalizacao.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_equalizacao.png
    (ver a versao Python)")
```

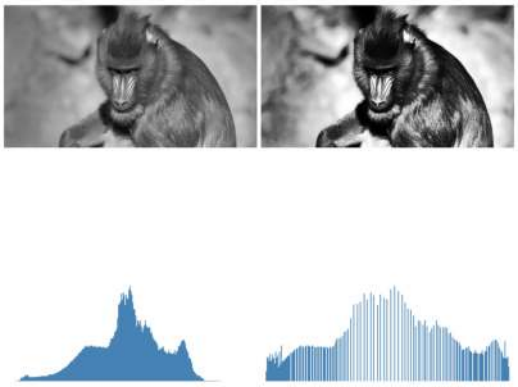


Figure 3.8: Equalização de histograma global (mm::equalize, via CDF) e os histogramas antes/depois. CLAHE (adaptativa) fica só na trilha Python — não tem equivalente em morph.hpp.

3.3.2 Histogram Specification

While equalization imposes a uniform distribution, **histogram specification** (*histogram matching*) allows the output image's histogram to follow an **arbitrary** distribution — for example, the histogram of another reference image.

The procedure involves three steps:

1. Compute the CDF of the input image: $P_r(r_k)$.
2. Compute the CDF of the reference image: $P_z(z_k)$.
3. For each level r_k, find the level z that minimizes $|P_z(z) - P_r(r_k)|$.

$$T(r_k) = \arg\min_z |P_z(z) - P_r(r_k)| \tag{3.8}$$

In Figure 3.9, we transfer the tonal profile of the leopard (Figure 3.3) to the mandrill image — a direct application of the concept seen in *blending*: instead of blending pixels, here we blend tonal distributions.

```
%%writefile tmp/fig_03_especificacao.cpp
#define MM_OUT "tmp/fig_03_especificacao.png"
//| label: fig-03-especificacao
//| fig-cap: "Especificação de histograma (mapear o mandril para o perfil tonal do leopardo)
precisa da CDF inversa da referência - fica só na trilha Python. Aqui, a equalização global do
mandril, como comparação."
//| echo: true
//| output: true

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_8.png");
mm::Image img_leop_gray = mm::_read_state("tmp/state/img_leop_gray_12.png");
// [pdi:state-io:end]
```

```
    // img_gray and img_leop_gray are pre-initialized

    mm::Image img_eq = mm::equalize(img_gray);

    mm::show({img_gray, img_leop_gray, img_eq}, MM_OUT,
             {"Mandril (original)", "Leopardo (referencia)", "Mandril equalizado"}, 3);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray, "tmp/fig_03_especificacao_0.png");
mm::write(img_leop_gray, "tmp/fig_03_especificacao_1.png");
mm::write(img_eq, "tmp/fig_03_especificacao_2.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_03_especificacao.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_especificacao.cpp -o tmp/fig_03_especificacao \
  && ./tmp/fig_03_especificacao \
  && test -f "tmp/fig_03_especificacao.png" \
  || echo "  mm::show não gravou tmp/fig_03_especificacao.png"
```

```
[1] Mandril (original)
[2] Leopardo (referencia)
[3] Mandril equalizado
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_03_especificacao_0.png"),
            mm.read("tmp/fig_03_especificacao_1.png"),
            mm.read("tmp/fig_03_especificacao_2.png"),
        ],
        titles=[
            'Mandril (original)',
            'Leopardo (referencia)',
            'Mandril equalizado',
        ],
        cols=3,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_03_especificacao_0.png (ver a versao Python)")
```

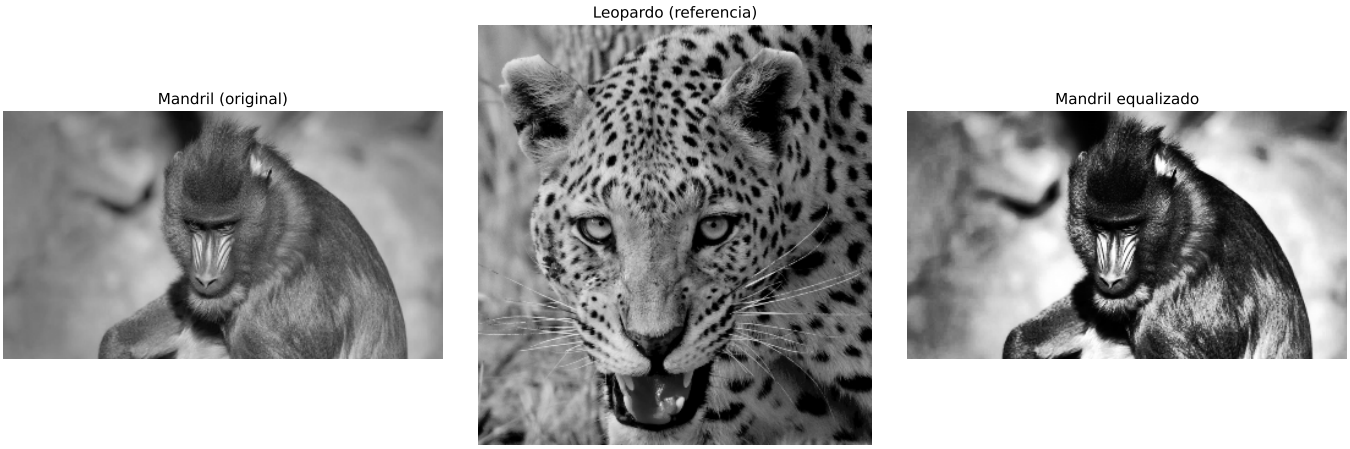


Figure 3.9: Especificação de histograma (mapear o mandril para o perfil tonal do leopardo) precisa da CDF inversa da referência — fica só na trilha Python. Aqui, a equalização global do mandril, como comparação.

3.4 Spatial Fundamentals: Neighborhood, Convolution, and *Kernels*

Spatial filtering operations do not act on a single isolated pixel, but on a **neighborhood** around it. For this purpose, a small matrix of coefficients called a **kernel** (or mask) is used, which traverses the entire image by means of a **sliding window**.

The most common windows are 3×3, 5×5, and 7×7. In a 3×3 window, for example, the central pixel is processed together with its eight immediate neighbors. At each window position, the pixel values are combined with the kernel coefficients, producing a new value for the central pixel.

3.4.1 Neighborhood

Consider a 3×3 window centered on the pixel (x, y):

$$\begin{bmatrix} (x-1, y-1) & (x, y-1) & (x+1, y-1) \\ (x-1, y) & (x, y) & (x+1, y) \\ (x-1, y+1) & (x, y+1) & (x+1, y+1) \end{bmatrix} \tag{3.9}$$

In general, a window of size $(2a+1) \times (2b+1)$ encompasses all pixels located up to a positions horizontally and up to b positions vertically relative to the central pixel. Thus, a 3×3 window corresponds to $a = b = 1$, a 5×5 window to $a = b = 2$, and so on.

Mathematically, the neighborhood is defined by

$$\mathcal{V}(x, y) = \{(x+s,\, y+t) : -a \leq s \leq a,\, -b \leq t \leq b\} \tag{3.10}$$

3.4.2 Edge Handling

Pixels near the edges have part of their neighborhood outside the image. To apply filters in these regions, it is necessary to define how the external values will be obtained. The three most common strategies (with the equivalent OpenCV constant in parentheses) are:

- **Zero-padding** (`BORDER_CONSTANT`): fills the external region with zeros.
- **Replication** (`BORDER_REPLICATE`): repeats the value of the border pixel.
- **Reflection** (`BORDER_REFLECT_101`): mirrors the neighboring pixels, without repeating the border pixel.

`mm::conv` uses reflection by default, as it better preserves the continuity of gray levels and reduces artifacts in edge handling.

The following example compares the three strategies with `mm::pad` on a 3×3 matrix. Note how each one fills the external pixels needed to apply a 3×3 filter also at the corners.

```
%%writefile tmp/mm_out_1.cpp
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>

int main() {
    mm::Image img(3, 3);
    img.at(0, 0) = 1; img.at(0, 1) = 2; img.at(0, 2) = 3;
    img.at(1, 0) = 4; img.at(1, 1) = 5; img.at(1, 2) = 6;
    img.at(2, 0) = 7; img.at(2, 1) = 8; img.at(2, 2) = 9;

    std::vector<std::string> bordas = {"constant", "replicate", "reflect101"};

    std::cout << "Imagem original:\n";
    std::cout << mm::drawImg(img) << "\n";

    for (int k : {3, 5}) {
        int b = k / 2;
        std::cout << "=== Kernel " << k << "x" << k << " (padding b=" << b << ") ===\n";
        for (const std::string& nome : bordas) {
            std::cout << nome << "\n";

            mm::Border border;
            if (nome == "constant") border = mm::Border::CONSTANT;
            else if (nome == "replicate") border = mm::Border::REPLICATE;
            else border = mm::Border::REFLECT101;

            std::cout << mm::drawImg(mm::pad(img, b, border)) << "\n";
        }
    }

    return 0;
}
```

```
Overwriting tmp/mm_out_1.cpp
```

```
!g++ -I. -std=c++17 tmp/mm_out_1.cpp -o tmp/mm_out_1 \
  && ./tmp/mm_out_1
```

```
Imagem original:
1 2 3
4 5 6
7 8 9

=== Kernel 3x3 (padding b=1) ===
constant
0 0 0 0 0
0 1 2 3 0
0 4 5 6 0
0 7 8 9 0
0 0 0 0 0

replicate
1 1 2 3 3
1 1 2 3 3
4 4 5 6 6
```

```
7 7 8 9 9
7 7 8 9 9

reflect101
5 4 5 6 5
2 1 2 3 2
5 4 5 6 5
8 7 8 9 8
5 4 5 6 5

=== Kernel 5x5 (padding b=2) ===
constant
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 1 2 3 0 0
0 0 4 5 6 0 0
0 0 7 8 9 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0

replicate
1 1 1 2 3 3 3
1 1 1 2 3 3 3
1 1 1 2 3 3 3
4 4 4 5 6 6 6
7 7 7 8 9 9 9
7 7 7 8 9 9 9
7 7 7 8 9 9 9

reflect101
9 8 7 8 9 8 7
6 5 4 5 6 5 4
3 2 1 2 3 2 1
6 5 4 5 6 5 4
9 8 7 8 9 8 7
6 5 4 5 6 5 4
3 2 1 2 3 2 1
```

Note that the result of a filter can vary significantly depending on the treatment adopted for the image borders.

In `morph.hpp`, the filtering functions (`mm::conv`, `mm::blur`, `mm::gaussian`, `mm::laplacian`, `mm::usm`) apply *padding* by **reflection** (`mm::Border::REFLECT101`) by default — the same behavior as `cv2.filter2D`. The didactic variant `mm::conv0` uses `mm::Border::KEEP`: the border pixels retain their original value, without the filter. Meanwhile, `mm::sobel` and `mm::prewitt` leave the border at zero (they compute only the interior).

3.4.3 Correlation vs. Convolution

There are two mathematically related mechanisms.

Cross-correlation — the *kernel* is applied directly:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)\, f(x+s,\, y+t) \tag{3.11}$$

Two-dimensional convolution — the *kernel* is rotated 180° before application:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)\, f(x-s,\, y-t) \tag{3.12}$$

For symmetric *kernels* (Gaussian, Laplacian, mean), the two operations yield identical results. For asymmetric *kernels* (Sobel, Prewitt), the difference is significant, as shown in the following examples.

3.4.3.1 Correlation (`mm::conv`)

```
%%writefile tmp/mm_out_2.cpp
#include "morph.hpp"
#include <iostream>

int main() {
    // imagem 4x4 (uint8) e kernel assimétrico 3x3
    mm::Image img(4, 4);
    // fill with values 1..16
    {
        int vals[4][4] = {{ 1,  2,  3,  4},
                          { 5,  6,  7,  8},
                          { 9, 10, 11, 12},
                          {13, 14, 15, 16}};
        for (int y = 0; y < 4; ++y)
            for (int x = 0; x < 4; ++x)
                img.at(y, x) = vals[y][x];
    }

    mm::Kernel w{{0, 1, 2},
                 {0, 0, 0},
                 {0, 0, 0}};

    mm::Image corr = mm::conv(img, w, mm::Border::CONSTANT);  // zero fora da imagem

    std::cout << "Imagem original:\n";          std::cout << mm::drawImg(img) << "\n";
    std::cout << "Kernel:\n";                   std::cout << mm::drawImg(w) << "\n";
    std::cout << "Resultado da correlação:\n"; std::cout << mm::drawImg(corr) << "\n";

    return 0;
}
```

```
Overwriting tmp/mm_out_2.cpp
```

```
!g++ -I. -std=c++17 tmp/mm_out_2.cpp -o tmp/mm_out_2 \
  && ./tmp/mm_out_2
```

```
Imagem original:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16

Kernel:
   0    1    2
   0    0    0
   0    0    0

Resultado da correlação:
 0  0  0  0
 5  8 11  4
17 20 23  8
29 32 35 12
```

`mm::conv` performs correlation, that is, it applies the *kernel* exactly in the provided orientation.

3.4.3.2 Convolution

```
%%writefile tmp/mm_out_3.cpp
// Compile with: g++ -std=c++17 -I. -o program program.cpp -lprotobuf -pthread
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>

int main() {
    // repeated here so the cell is self-contained
    mm::Image img(4, 4);
    // Initialize img with values 1..16
    for (int y = 0; y < img.h; y++) {
        for (int x = 0; x < img.w; x++) {
            img.at(y, x) = static_cast<unsigned char>(y * 4 + x + 1);
        }
    }

    mm::Kernel w{{0,1,2},{0,0,0},{0,0,0}};

    // kernel rotated 180° (equivalent to np.rot90(w, 2))
    mm::Kernel w_conv{{0,0,0},{0,0,0},{2,1,0}};

    mm::Image conv_result = mm::conv(img, w_conv, mm::Border::CONSTANT);

    std::cout << "Imagem original:";            std::cout << mm::drawImg(img) << "\n";
    std::cout << "Kernel original:";            std::cout << mm::drawImg(w) << "\n";
    std::cout << "Kernel rotacionado 180°:";    std::cout << mm::drawImg(w_conv) << "\n";
    std::cout << "Resultado da convolução:";    std::cout << mm::drawImg(conv_result) << "\n";

    return 0;
}
```

```
Overwriting tmp/mm_out_3.cpp
```

```
!g++ -I. -std=c++17 tmp/mm_out_3.cpp -o tmp/mm_out_3 \
  && ./tmp/mm_out_3
```

```
Imagem original: 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16

Kernel original:   0    1    2
   0    0    0
   0    0    0

Kernel rotacionado 180°:   0    0    0
   0    0    0
   2    1    0

Resultado da convolução: 5 16 19 22
 9 28 31 34
13 40 43 46
 0  0  0  0
```

The convolution uses the *kernel* rotated by 180°. To reproduce the mathematical definition of convolution, the *kernel* is rotated (here, `[[0,1,2],[0,0,0],[0,0,0]]` → `[[0,0,0],[0,0,0],[2,1,0]]`) before applying `mm::conv`.

3.4.4 The Role of the *Kernel*

The *kernel* coefficients completely determine the effect produced by the filter, as summarized in Table 3.2.

Table 3.2: Typical interpretation of *kernel* coefficients.

Feature	Typical effect
Positive coefficients summing to 1	Smoothing (*low-pass*)
Sum equal to 0, with positive and negative values	Edge detection (*high-pass*)
Dominant central positive coefficient and negative neighbors	Sharpening
Asymmetric coefficients	Directional gradient

Examples:

Smoothing:

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

Edge detection:

$$\begin{bmatrix}-1 & -1 & -1\\ -1 & 8 & -1\\ -1 & -1 & -1\end{bmatrix}$$

Sharpening:

$$\begin{bmatrix}0 & -1 & 0\\ -1 & 5 & -1\\ 0 & -1 & 0\end{bmatrix}$$

Directional gradient (Sobel):

$$\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix}$$

Figure 3.10 demonstrates the step-by-step mechanism: for each position of the window, each coefficient of the *kernel* is multiplied by the corresponding pixel in the neighborhood, and the resulting products are summed. The result is exactly the value defined by Equation 3.11 for that position in the image. Although the images produced by `mm::conv0` and `cv2.filter2D` (or `mm::conv`) are visually very similar, the OpenCV-based implementation is thousands of times faster, as shown below.

⚠ Performance: Python loops vs. vectorized operations

The `mm::conv0` function implements correlation directly in Python through nested loops. Although this approach is suitable for didactic purposes, it executes a large number of operations and becomes slow for larger images.
In contrast, `mm::conv` uses `cv2.filter2D`, implemented in C++ and optimized for matrix operations. In the example presented, the vectorized version was more than **3000 times faster** than the didactic implementation, producing a visually equivalent result.
The numerical differences observed are mainly concentrated at the image borders. In `mm::conv0`, border pixels remain unchanged, while `mm::conv` uses a border reflection strategy (`cv2.BORDER_REFLECT_101`, the default for `cv2.filter2D`).
Therefore, `mm::conv0` should be used to understand the algorithm, while `mm::conv` is the recommended option for practical applications.

```
%%writefile tmp/fig_03_convolucao_passo.cpp
#define MM_OUT "tmp/fig_03_convolucao_passo.png"
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_leop_gray = mm::_read_state("tmp/state/img_leop_gray_12.png");
// [pdi:state-io:end]

    //#| label: fig-03-convolucao-passo
    //#| fig-cap: "Correlação com *kernel* de média 3×3: versão didática mm::conv0 (bordas
    preservadas) vs. mm::conv (borda refletida). A diferença se concentra nas bordas."
    //#| echo: true
    //#| output: true

    // w_mean = np.ones((3, 3), dtype=np.float32) / 9.0
    mm::Kernel w_mean = mm::Kernel::mean(3);

    // img_gray = img_leop_gray
    mm::Image img_gray = img_leop_gray;

    // img_conv0 = mm.conv0(img_gray, w_mean)   # loops, borders preserved
    mm::Image img_conv0 = mm::conv0(img_gray, w_mean);

    // img_conv = mm.conv(img_gray, w_mean)    # reflected border
    mm::Image img_conv = mm::conv(img_gray, w_mean);

    std::cout << "Correlacao no pixel central [251,251]:" << std::endl;
    std::cout << "  original = " << (int)img_gray.at(251, 251) << std::endl;
    std::cout << "  conv0    = " << (int)img_conv0.at(251, 251) << std::endl;
    std::cout << "  conv     = " << (int)img_conv.at(251, 251) << std::endl;

    mm::show(std::vector<mm::Image>{img_gray, img_conv0, img_conv},
             MM_OUT,
             std::vector<std::string>{"Original", "conv0 (laços)", "conv (vetorizado)"}, 3);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray, "tmp/fig_03_convolucao_passo_0.png");
mm::write(img_conv0, "tmp/fig_03_convolucao_passo_1.png");
mm::write(img_conv, "tmp/fig_03_convolucao_passo_2.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_03_convolucao_passo.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_convolucao_passo.cpp -o tmp/fig_03_convolucao_passo \
  && ./tmp/fig_03_convolucao_passo \
  && test -f "tmp/fig_03_convolucao_passo.png" \
  || echo "  mm::show não gravou tmp/fig_03_convolucao_passo.png"
```

```
Correlacao no pixel central [251,251]:
  original = 104
  conv0    = 102
  conv     = 102
```

```
[1] Original
[2] conv0 (laços)
[3] conv (vetorizado)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_03_convolucao_passo_0.png"),
            mm.read("tmp/fig_03_convolucao_passo_1.png"),
            mm.read("tmp/fig_03_convolucao_passo_2.png"),
        ],
        titles=[
            'Original',
            'conv0 (laços)',
            'conv (vetorizado)',
        ],
        cols=3,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_03_convolucao_passo_0.png (ver a versao Python)")
```

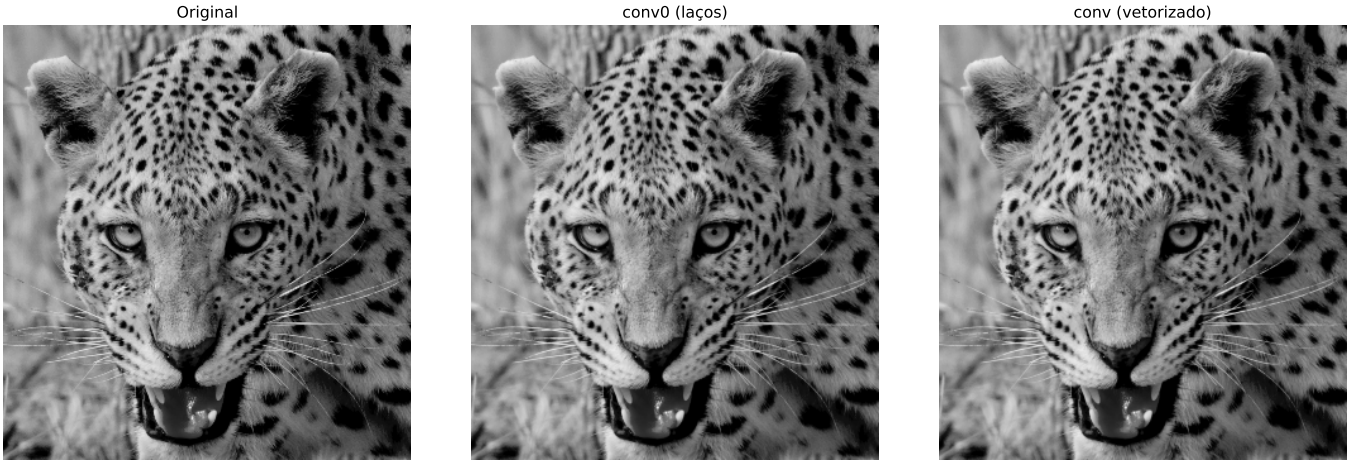


Figure 3.10: Correlação com *kernel* de média 3×3: versão didática mm::conv0 (bordas preservadas) vs. mm::conv (borda refletida). A diferença se concentra nas bordas.

```
%%writefile tmp/mm_out_4.cpp
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <filesystem>

//| echo: false
// From here on the "subject" of the filtering examples becomes the
// leopard (more texture and edges than the mandrill).
int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_leop = mm::_read_state("tmp/state/img_leop_12.png");
// [pdi:state-io:end]

    mm::Image img_gray = mm::gray(img_leop);

// [pdi:state-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp/state");
```

```
mm::write(img_gray, "tmp/state/img_gray_45.png");
// [pdi:state-io:end]
return 0;
}
```

```
Overwriting tmp/mm_out_4.cpp
```

```
!g++ -I. -std=c++17 tmp/mm_out_4.cpp -o tmp/mm_out_4 \
  && ./tmp/mm_out_4
```

3.4.5 Numerical Example: Step-by-Step Correlation

To make the mechanism of Equation 3.11 concrete, consider the 3×3 averaging *kernel* ($a = b = 1$, all coefficients $= 1/9 \approx 0.111$) applied to the 5×5 *patch* extracted from the leopard image. Figure 3.11 displays the *patch* with a grid and highlights in yellow the 3×3 window centered at pixel $[1, 1]$:

```
%%writefile tmp/fig_03_patch.cpp
#define MM_OUT "tmp/fig_03_patch.png"
#include "morph.hpp"
#include <iostream>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_45.png");
// [pdi:state-io:end]

    //| label: fig-03-patch
    //| fig-cap: "*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."
    //| echo: true
    //| output: true

    mm::Image patch = mm::crop(img_gray, 250, 255, 250, 255);
    mm::Kernel B{{1,1,1},{1,1,1},{1,1,1}};

    std::cout << "Patch 5×5 (intensidades):\n";
    std::cout << mm::drawImg(patch);

    mm::drawImgKernel(patch, B, 1, 1, MM_OUT, 40);

    return 0;
}
```

```
Overwriting tmp/fig_03_patch.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_patch.cpp -o tmp/fig_03_patch \
  && ./tmp/fig_03_patch \
  && test -f "tmp/fig_03_patch.png" \
  || echo "  mm::show não gravou tmp/fig_03_patch.png"
```

```
Patch 5×5 (intensidades):
 94  96 103 113 115
107 104 103 107 114
 98 102 112 117 116
 81  91 112 122 118
 85  91 110 119 121
Processando pixel (x,y)=(1,1)  |  janela do kernel 3x3
```

```
 94  96 103 113 115
107 104 103 107 114
 98 102 112 117 116
 81  91 112 122 118
 85  91 110 119 121
```

```
try:
    mm.show(mm.read("tmp/fig_03_patch.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_patch.png (ver
    a versao Python)")
```

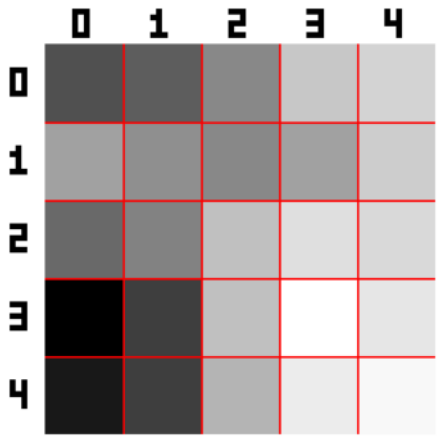


Figure 3.11: *Patch* 5×5 extraído da imagem do leopardo (posição [250:255, 250:255]). A janela amarela destaca a vizinhança 3×3 centrada no pixel [1,1] onde a correlação será calculada.

To illustrate the calculation of correlation, consider the pixel at position $[1, 1]$ in the 5×5 *patch* shown in Figure 3.11. This position was chosen solely for didactic convenience, as it has a complete 3×3 neighborhood around it.

The values in this neighborhood correspond to the upper-left submatrix of the *patch*:

$$\text{neighborhood} = \begin{bmatrix} 91 & 95 & 108 \\ 106 & 107 & 108 \\ 102 & 103 & 107 \end{bmatrix}$$

Applying Equation 3.11 with the mean kernel:

$$g[1,1] = \frac{91 + 95 + 108 + 106 + 107 + 108 + 102 + 103 + 107}{9} = \frac{927}{9} = 103$$

The result (103) is slightly lower than the original value of the central pixel (107), because the mean incorporates neighbors of lower intensity, producing a smoothing effect. In practice, the algorithm starts processing at $[0, 0]$ and repeats this same calculation for each position of the image, shifting the window until it covers the entire domain.

3.5 Smoothing Spatial Filtering

Smoothing filters attenuate abrupt intensity variations, reducing noise and high-frequency details. They are **low-pass** filters — they preserve low-frequency components (large structures) and attenuate high-frequency ones (noise, edges).

3.5.1 Mean Filter (*Box Filter*)

The mean filter uses a uniform kernel of size $n \times n$, where all coefficients equal $1/n^2$:

$$w_{\text{mean}} = \frac{1}{n^2} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}_{n \times n} \tag{3.13}$$

Each output pixel is the arithmetic mean of the n^2 pixels in its neighborhood. Note that the sum of the coefficients is always 1 — the average brightness of the image is preserved. Larger kernels produce more aggressive smoothing but progressively blur the edges.

Figure 3.12 shows the effect of the mean filter with 3×3, 7×7, and 15×15 kernels on a detail of the leopard image. The results were obtained with `mm::blur`, which implements the mean filter using the `cv2.blur` function, equivalent to convolving the image with a uniform kernel whose coefficients are $h(x, y) = 1/N^2$; equivalently, the same result can be obtained with `mm::conv`, computing $(g = f * h)$. As the kernel size increases, more pixels contribute to each output value, intensifying smoothing, reducing noise, and making fine details and edges progressively more blurred.

```
%%writefile tmp/fig_03_media.cpp
#define MM_OUT "tmp/fig_03_media.png"
//| label: fig-03-media
//| fig-cap: "Filtro de média com *kernels* de tamanho crescente (3×3, 7×7, 15×15). O
borramento das bordas aumenta com o tamanho do *kernel*."
//| echo: true
//| output: true

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_45.png");
// [pdi:state-io:end]

    // Detalhe da região do olho
    int y0 = 580, y1 = 740, x0 = 680, x1 = 900;
    mm::Image img_gray_crop = mm::crop(img_gray, y0, y1, x0, x1);

    std::vector<int> sizes = {3, 7, 15};
    std::vector<mm::Image> imgs = {img_gray_crop};
    for (int k : sizes) {
        imgs.push_back(mm::blur(img_gray_crop, k));
    }
    std::vector<std::string> titles = {"Original"};
    for (int k : sizes) {
        titles.push_back("Média " + std::to_string(k) + "×" + std::to_string(k));
    }

    mm::show(imgs, MM_OUT, titles, 4);

    return 0;
}
```

```
Overwriting tmp/fig_03_media.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_media.cpp -o tmp/fig_03_media \
  && ./tmp/fig_03_media \
  && test -f "tmp/fig_03_media.png" \
  || echo "  mm::show não gravou tmp/fig_03_media.png"
```

```
[1] Original
[2] Média 3×3
[3] Média 7×7
[4] Média 15×15
```

```
try:
    mm.show(mm.read("tmp/fig_03_media.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_media.png (ver
    a versao Python)")
```

Figure 3.12: Filtro de média com *kernels* de tamanho crescente (3×3, 7×7, 15×15). O borramento das bordas aumenta com o tamanho do *kernel*.

3.5.2 Gaussian Filter

The Gaussian filter weights the pixels in the neighborhood according to a two-dimensional Gaussian function:

$$G(s,t) = \frac{1}{2\pi\sigma^2}\, e^{-\frac{s^2+t^2}{2\sigma^2}} \tag{3.14}$$

where σ is the standard deviation and controls the radius of influence. Pixels closer to the center have greater weight; distant pixels are progressively ignored.

Figure 3.13 presents the Gaussian kernel 5×5 generated for $\sigma = 1$. The kernel was constructed from the outer product of two one-dimensional Gaussian vectors and subsequently normalized so that the sum of its coefficients equals 1. It is observed that the largest weights are concentrated at the center of the matrix, decreasing radially toward the edges. This distribution causes the central pixels to have greater influence on the filtering result, contributing to a more natural smoothing with better edge preservation than the mean filter.

```
%%writefile tmp/fig_03_gauss_kernel.cpp
#define MM_OUT "tmp/fig_03_gauss_kernel.png"
// Compile: g++ -std=c++17 -o program program.cpp -I. -L. -lmorph -lopencv_core
-lopencv_imgproc -lopencv_imgcodecs
#include "morph.hpp"
#include <iostream>
#include <iomanip>

int main() {
    //| label: fig-03-gauss-kernel
```

```
    //| fig-cap: "*Kernel* Gaussiano 5×5 ( =1): resposta ao impulso do filtro - pesos maiores
    no centro, decrescendo radialmente."
    //| echo: true
    //| output: true

    mm::Kernel w = mm::Kernel::gaussian(5, 1.0);   // mm::Kernel::gaussian(5, 1.0) na
morph.hpp

    std::cout << "Kernel Gaussiano 5x5 (s=1), normalizado:\n";
    for (int y = 0; y < 5; y++) {
        std::cout << "  ";
        for (int x = 0; x < 5; x++) {
            std::cout << std::fixed << std::setprecision(4) << w.at(y, x) << "  ";
        }
        std::cout << "\n";
    }
    std::cout << "Peso central [2,2] = " << std::fixed << std::setprecision(4) << w.at(2, 2)
              << "   |   canto [0,0] = " << std::fixed << std::setprecision(4) << w.at(0, 0)
              << "\n";

    // Visualização: resposta do filtro Gaussiano a um impulso central
    mm::Image impulso(5, 5);
    impulso.at(2, 2) = 255;
    mm::drawImgPlt(mm::gaussian(impulso, 5, 1.0), MM_OUT);

    return 0;
}
```

```
Overwriting tmp/fig_03_gauss_kernel.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_gauss_kernel.cpp -o tmp/fig_03_gauss_kernel \
  && ./tmp/fig_03_gauss_kernel \
  && test -f "tmp/fig_03_gauss_kernel.png" \
  || echo "  mm::show não gravou tmp/fig_03_gauss_kernel.png"
```

```
Kernel Gaussiano 5x5 (s=1), normalizado:
  0.0030  0.0133  0.0219  0.0133  0.0030
  0.0133  0.0596  0.0983  0.0596  0.0133
  0.0219  0.0983  0.1621  0.0983  0.0219
  0.0133  0.0596  0.0983  0.0596  0.0133
  0.0030  0.0133  0.0219  0.0133  0.0030
Peso central [2,2] = 0.1621   |   canto [0,0] = 0.0030
 3  7 11  7  3
 7 15 25 15  7
11 25 41 25 11
 7 15 25 15  7
 3  7 11  7  3
```

```
try:
    mm.show(mm.read("tmp/fig_03_gauss_kernel.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_03_gauss_kernel.png (ver a versao Python)")
```

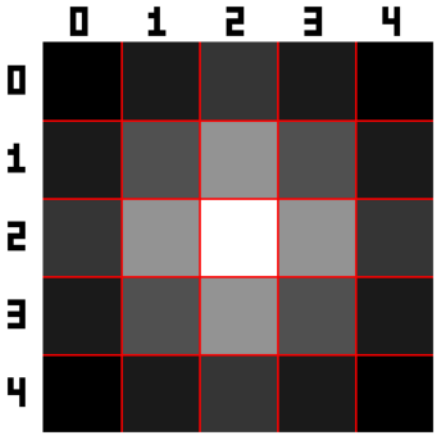


Figure 3.13: *Kernel* Gaussiano 5×5 (=1): resposta ao impulso do filtro — pesos maiores no centro, decrescendo radialmente.

i Computational advantage of separability

Consider a square *kernel* of size $n \times n$. If this filter is separable (such as the Gaussian), the 2D convolution can be decomposed into two 1D convolutions: one horizontal and one vertical.
In this case, the cost per pixel decreases from approximately $O(n^2)$ operations (direct 2D convolution) to $O(2n)$ operations (two 1D convolutions). Thus, the complexity is significantly reduced, making processing more efficient

Compared to the averaging filter, the Gaussian:

- **Better preserves edges** — the radial weighting smooths without creating abrupt transitions;
- **Does not introduce ringing** in the frequency domain, since the Gaussian is its own Fourier transform (Chapter 5);
- **Is controlled by** σ — increasing σ is equivalent to increasing the smoothing radius in a continuous and predictable manner.

Figure 3.14 compares the mean and Gaussian filters applied to the leopard image using a 9×9 window. The mean filter was implemented by convolution with a uniform *kernel*, where all 81 pixels in the neighborhood have the same weight (1/81), while the Gaussian filter was obtained with `cv2.GaussianBlur`, using weights defined by a Gaussian distribution. Both reduce noise and smooth the image, but the Gaussian filter better preserves edges and local details, as can be observed in the enlarged region of the eye.

Figure 3.14 compares the mean and Gaussian filters applied to a detail of the leopard image with 9×9 *kernels*. The mean filter was obtained with `mm::blur`, equivalent to convolution with a uniform *kernel* whose coefficients equal 1/81, while the Gaussian filter was obtained with `mm::gaussian`, equivalent to convolution with a *kernel* generated from a Gaussian distribution. Both promote smoothing and noise reduction, but the Gaussian filter assigns greater weight to the central pixels of the neighborhood, better preserving edges and local details, as can be observed in the enlarged region of the eye.

```
%%writefile tmp/fig_03_gauss.cpp
#define MM_OUT "tmp/fig_03_gauss.png"
// C++ translation
// Compile: g++ -std=c++17 -o program program.cpp -I<path-to-morph.hpp>

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>
```

```
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_45.png");
// [pdi:state-io:end]

    // #| label: fig-03-gauss
    // #| fig-cap: "Comparação entre filtro de média e Gaussiano (*kernel* 9×9, =0). O
    Gaussiano preserva melhor as bordas, visível no detalhe do rosto."
    // #| echo: true
    // #| output: true

    mm::Image img_media9 = mm::blur(img_gray, 9);
    mm::Image img_gauss9 = mm::gaussian(img_gray, 9, 0);

    // Detalhe da região do olho
    mm::Image img_gray_crop = mm::crop(img_gray, 580, 740, 680, 900);
    mm::Image img_media9_crop = mm::crop(img_media9, 580, 740, 680, 900);
    mm::Image img_gauss9_crop = mm::crop(img_gauss9, 580, 740, 680, 900);

    mm::show(
        std::vector<mm::Image>{img_gray_crop, img_media9_crop, img_gauss9_crop},
        MM_OUT,
        std::vector<std::string>{"Detalhe: Original", "Média 9×9", "Gaussiano 9×9"},
        3
    );


// [pdi:state-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp/state");
mm::write(img_gray_crop, "tmp/state/img_gray_crop_58.png");
// [pdi:state-io:end]

// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray_crop, "tmp/fig_03_gauss_0.png");
mm::write(img_media9_crop, "tmp/fig_03_gauss_1.png");
mm::write(img_gauss9_crop, "tmp/fig_03_gauss_2.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_03_gauss.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_gauss.cpp -o tmp/fig_03_gauss \
  && ./tmp/fig_03_gauss \
  && test -f "tmp/fig_03_gauss.png" \
  || echo "  mm::show não gravou tmp/fig_03_gauss.png"
```

```
[1] Detalhe: Original
[2] Média 9×9
[3] Gaussiano 9×9
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_03_gauss_0.png"),
            mm.read("tmp/fig_03_gauss_1.png"),
            mm.read("tmp/fig_03_gauss_2.png"),
```

```
        ],
        titles=[
            'Detalhe: Original',
            'Média 9×9',
            'Gaussiano 9×9',
        ],
        cols=3,
        figsize=(12, 8),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_gauss_0.png
    (ver a versao Python)")
```

Figure 3.14: Comparação entre filtro de média e Gaussiano (*kernel* 9×9, =0). O Gaussiano preserva melhor as bordas, visível no detalhe do rosto.

3.6 Enhancement Spatial Filtering

Enhancement filters emphasize abrupt intensity transitions, increasing sharpness and edge visibility. They are **high-pass** filters — they amplify high-frequency components (edges, texture) and suppress low-frequency ones (uniform regions).

The intuition is simple: if we subtract from an image its smoothed version (which contains only low frequencies), what remains are the high frequencies — edges and details. Adding this residual back to the original image increases local contrast:

$$g = f + k\,(f - f_{\text{smooth}}), \quad k > 0 \tag{3.15}$$

Figure 3.15 illustrates this process on a synthetic 1D signal with three distinct structures: a wide step, a narrow peak, and a smooth ramp. Panel shows the original signal $f(x)$; panel shows the smoothed version $f_{\text{smooth}}(x)$ obtained by moving average — note how the narrow peak is attenuated. Panel displays the residual $f - f_{\text{smooth}}$, which retains only the abrupt transitions. Finally, panel shows $g(x)$: the previously attenuated peak is restored and amplified relative to the original. Adjust k and the window size to observe the trade-off between sharpness and noise amplification.

Enhancement filters formalize this idea directly in the kernel, without requiring two separate steps.

3.6.1 Laplacian

The Laplacian is an isotropic **second-derivative** operator, meaning it responds equally to variations in all directions, unlike first-derivative operators such as Sobel and Prewitt, which are directional:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{3.16}$$

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap03/sim-03-filtragem1d.html

Figure 3.15: Simulator: 1D Spatial Enhancement Filtering (Unsharp Masking and High-Boost)

An important property of the second derivative is that its value is **close to zero in uniform regions** and high at intensity transitions. Thus, by subtracting the Laplacian from the original image, edges and details are enhanced, increasing local contrast:

$$g(x,y) = f(x,y) - \nabla^2 f(x,y) \tag{3.17}$$

In the discrete form, the second derivative in x is approximated by $f(x+1,y) - 2f(x,y) + f(x-1,y)$, and similarly in y. By summing the two directions, we obtain the *kernel* w_4 (4-neighbor) or w_8 (8-neighbor, including diagonals):

$$w_4 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \qquad w_8 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.18}$$

> **i** Zero sum and negative center
>
> Both *kernels* have **coefficient sum equal to zero**: in uniform regions, the output is 0 — the Laplacian does not alter the average brightness, it only detects variations. The **negative center** indicates that the pixel is compared with its neighbors: the more it stands out (upward or downward), the greater the absolute value of the Laplacian at that point.

In the following example, the central pixel $[1,1] = 107$ has neighbors $\{95, 106, 108, 103\}$. Since these values are close to one another, the region is nearly uniform, and the Laplacian returns a low value, producing little enhancement. In edge regions, where there are greater differences between the central pixel and its neighbors, the Laplacian assumes higher values (positive or negative), and the operation in Equation 3.17 intensifies these transitions.

Figure 3.16 illustrates the computation of the Laplacian using the *kernel* w_4 in a 3×3 neighborhood highlighted within a 5×5 patch. The example shows the value obtained by the operator and the corresponding enhanced pixel in the output image, which changes from 107 to 123.

```
%%writefile tmp/fig_03_laplaciano_patch.cpp
#define MM_OUT "tmp/fig_03_laplaciano_patch.png"
#include "morph.hpp"
#include <iostream>
#include <vector>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_45.png");
// [pdi:state-io:end]

    //| label: fig-03-laplaciano-patch
    //| fig-cap: "*Kernel* Laplaciano w4 sobre o *patch* 5×5: a janela amarela destaca a
    vizinhança 3×3 onde o operador de segunda derivada é calculado."
    //| echo: true
    //| output: true

    mm::Image patch = mm::crop(img_gray, 250, 255, 250, 255);

    std::cout << "Patch 5x5 (intensidades):" << std::endl;
    std::cout << mm::drawImg(patch) << std::endl;

    mm::Kernel B = {{1,1,1},{1,1,1},{1,1,1}};
    mm::drawImgKernel(patch, B, 1, 1, MM_OUT, 40);
```

```
    return 0;
}
```

```
Overwriting tmp/fig_03_laplaciano_patch.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_laplaciano_patch.cpp -o tmp/fig_03_laplaciano_patch \
  && ./tmp/fig_03_laplaciano_patch \
  && test -f "tmp/fig_03_laplaciano_patch.png" \
  || echo "  mm::show não gravou tmp/fig_03_laplaciano_patch.png"
```

```
Patch 5x5 (intensidades):
 94  96 103 113 115
107 104 103 107 114
 98 102 112 117 116
 81  91 112 122 118
 85  91 110 119 121

Processando pixel (x,y)=(1,1)  |  janela do kernel 3x3
 94  96 103 113 115
107 104 103 107 114
 98 102 112 117 116
 81  91 112 122 118
 85  91 110 119 121
```

```
try:
    mm.show(mm.read("tmp/fig_03_laplaciano_patch.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_03_laplaciano_patch.png (ver a versao Python)")
```

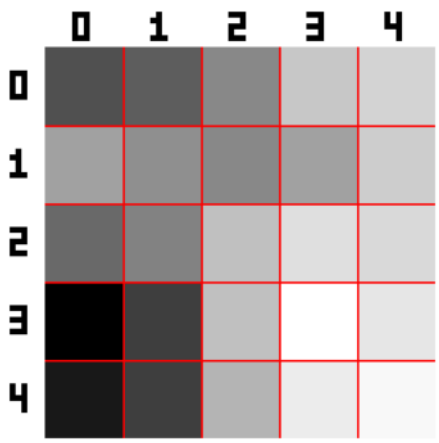


Figure 3.16: *Kernel* Laplaciano w4 sobre o *patch* 5×5: a janela amarela destaca a vizinhança 3×3 onde o operador de segunda derivada é calculado.

Figure 3.17 compares the application of the Laplacian *kernels* w_4 and w_8 on a larger crop of the leopard image. For each case, the raw operator response, which highlights edges and intensity transitions, and the image obtained after enhancement by Laplacian subtraction are shown. It is observed that the *kernel* w_8, by also considering diagonal neighbors, produces a stronger response and detects variations in more directions, resulting in a slightly more pronounced enhancement.

```
%%writefile tmp/fig_03_laplaciano.cpp
#define MM_OUT "tmp/fig_03_laplaciano.png"
```

```
//| label: fig-03-laplaciano
//| fig-cap: "Laplaciano applied to the leopard image: raw response (edges) with w4 and w8,
and images enhanced by subtracting the Laplacian. w8 is more sensitive to diagonals."
//| echo: true
//| output: true

#include "morph.hpp"
#include <vector>
#include <string>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray_crop = mm::_read_state("tmp/state/img_gray_crop_58.png");
// [pdi:state-io:end]

    mm::Kernel w4{{0,1,0},{1,-4,1},{0,1,0}};
    mm::Kernel w8{{1,1,1},{1,-8,1},{1,1,1}};

    mm::show(
        std::vector<mm::Image>{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)},
        MM_OUT,
        std::vector<std::string>{"Original", "Laplaciano w4", "Realce w4",
                "Original", "Laplaciano w8", "Realce w8"},
        3
    );

    return 0;
}
```

```
Overwriting tmp/fig_03_laplaciano.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_laplaciano.cpp -o tmp/fig_03_laplaciano \
  && ./tmp/fig_03_laplaciano \
  && test -f "tmp/fig_03_laplaciano.png" \
  || echo "  mm::show não gravou tmp/fig_03_laplaciano.png"
```

```
[1] Original
[2] Laplaciano w4
[3] Realce w4
[4] Original
[5] Laplaciano w8
[6] Realce w8
```

```
try:
    mm.show(mm.read("tmp/fig_03_laplaciano.png"), figsize=(14, 8))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_laplaciano.png
    (ver a versao Python)")
```

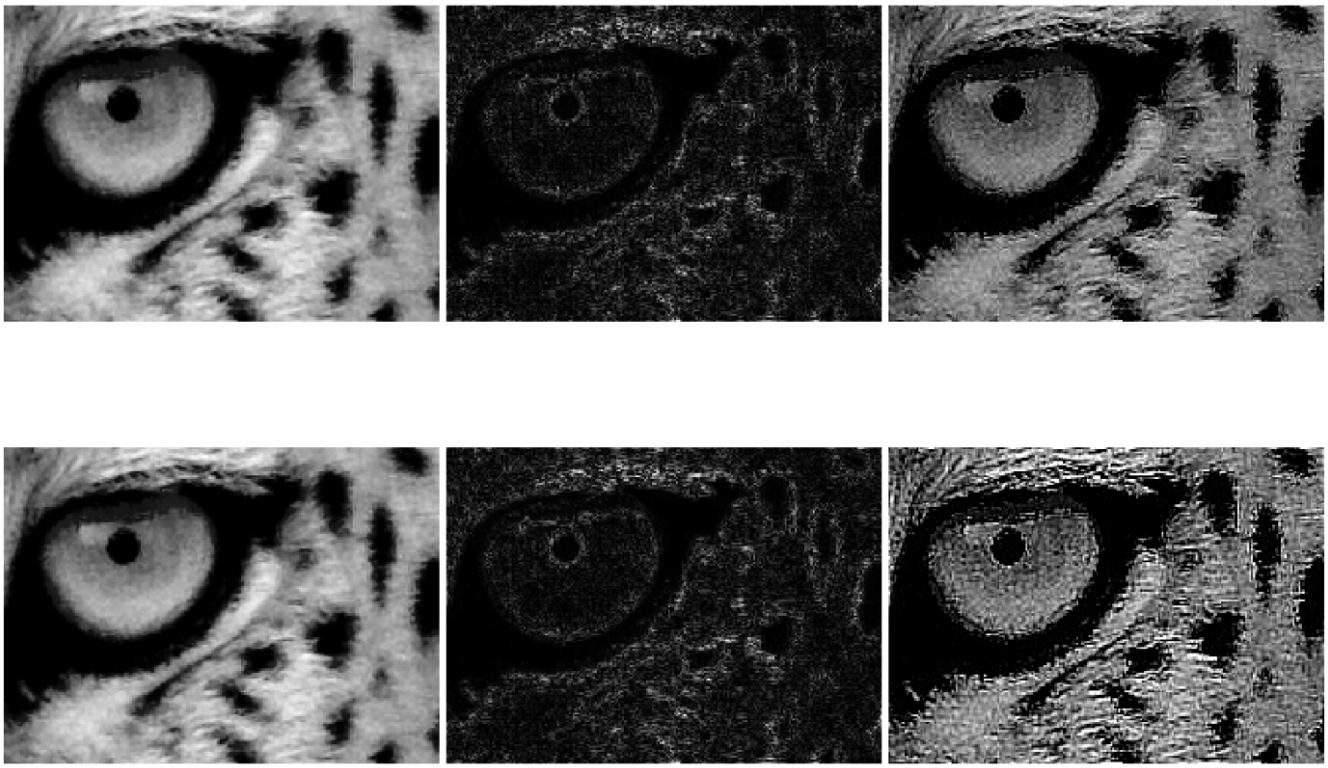


Figure 3.17: Laplaciano aplicado à imagem do leopardo: resposta bruta (bordas) com w4 e w8, e imagens realçadas pela subtração do Laplaciano. w8 é mais sensível às diagonais.

3.6.2 Sobel Operator

The Sobel operator estimates the **first-order partial derivatives** in the horizontal and vertical directions. Unlike the Laplacian (second derivative), Sobel is directional and more robust to noise, as each *kernel* combines a derivative with a perpendicular Gaussian smoothing:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * f, \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * f \tag{3.19}$$

G_x detects **vertical** edges (variation in the x direction); G_y detects **horizontal** edges (variation in the y direction). The weights $\{1, 2, 1\}$ in the perpendicular direction correspond to 1D Gaussian smoothing, which reduces sensitivity to noise.

> **i** Sobel is correlation, not convolution
>
> The Sobel *kernels* are **asymmetric** — a 180° rotation alters the result. `cv2.Sobel` implements cross-correlation (like `cv2.filter2D`). To obtain the correct directional derivative, the signs are already defined for correlation: G_x returns positive values where intensity increases from left to right.

The **gradient** magnitude combines the two components, representing edge strength independent of direction:

$$|\nabla f| = \sqrt{G_x^2 + G_y^2} \tag{3.20}$$

And the **direction** of the gradient (perpendicular to the edge) is:

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \tag{3.21}$$

To illustrate numerically, G_x and G_y are computed manually at the central pixel $[1, 1]$ of the 5×5 *patch*:

The reduced value of $|\nabla f|$ in this *patch* confirms that the region is nearly uniform, since the gradient assumes high values only where there are significant intensity changes. Figure 3.18 applies the Sobel operator to a larger crop of the leopard image. The horizontal (G_x) and vertical (G_y) responses are shown, obtained by convolution with the respective Sobel *kernels*, as well as the magnitude $|\nabla f|$, calculated from the combination of both. While G_x highlights vertical edges and G_y horizontal edges, the magnitude evidences edges in any direction.

```
%%writefile tmp/fig_03_sobel.cpp
#define MM_OUT "tmp/fig_03_sobel.png"
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray_crop = mm::_read_state("tmp/state/img_gray_crop_58.png");
// [pdi:state-io:end]

    //| label: fig-03-sobel
    //| fig-cap: "Operador de Sobel na imagem do leopardo: a magnitude | f| combina os
    gradientes horizontal e vertical, revelando todas as bordas. A decomposição Gx/Gy com
    sinal fica na trilha Python - mm::sobel devolve a magnitude já com clip."
    //| echo: true
    //| output: true

    mm::Image mag = mm::sobel(img_gray_crop);

    mm::show(std::vector<mm::Image>{img_gray_crop, mag},
             MM_OUT,
             std::vector<std::string>{"Original", "Magnitude |grad f| (mm.sobel)"}, 2);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray_crop, "tmp/fig_03_sobel_0.png");
mm::write(mag, "tmp/fig_03_sobel_1.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_03_sobel.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_sobel.cpp -o tmp/fig_03_sobel \
  && ./tmp/fig_03_sobel \
  && test -f "tmp/fig_03_sobel.png" \
  || echo "  mm::show não gravou tmp/fig_03_sobel.png"
```

```
[1] Original
[2] Magnitude |grad f| (mm.sobel)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_03_sobel_0.png"),
            mm.read("tmp/fig_03_sobel_1.png"),
        ],
        titles=[
            'Original',
            'Magnitude |grad f| (mm.sobel)',
        ],
        cols=2,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_sobel_0.png
    (ver a versao Python)")
```

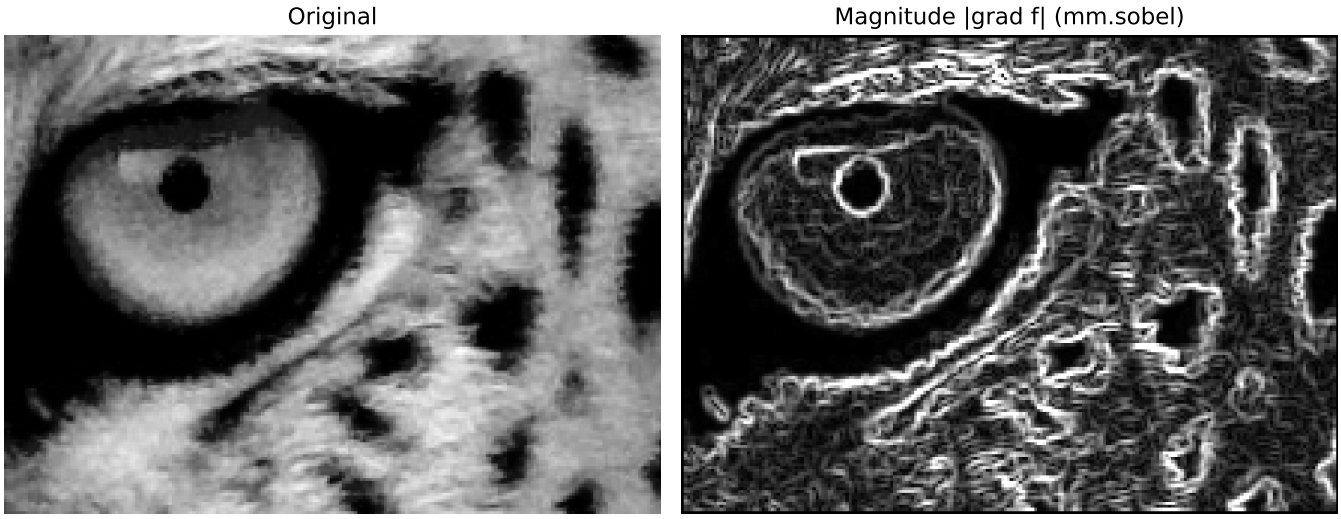


Figure 3.18: Operador de Sobel na imagem do leopardo: a magnitude | f| combina os gradientes horizontal e vertical, revelando todas as bordas. A decomposição Gx/Gy com sinal fica na trilha Python — mm::sobel devolve a magnitude já com clip.

3.6.3 Prewitt Operator

The Prewitt operator is structurally identical to Sobel, but replaces the Gaussian weighting $\{1, 2, 1\}$ with uniform weights $\{1, 1, 1\}$:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} * f, \qquad G_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} * f \tag{3.22}$$

The gradient magnitude and direction follow the same equations as Sobel (Equation 3.20 and Equation 3.21). The practical difference is that Prewitt is slightly more sensitive to noise—the uniform perpendicular smoothing weights the central pixel of the line less—but it is computationally simpler. In low-noise images, the results are equivalent.

```
%%writefile tmp/fig_03_prewitt.cpp
#define MM_OUT "tmp/fig_03_prewitt.png"
// Compile: g++ -std=c++17 -o prewitt prewitt.cpp $(pkg-config --cflags --libs morph)
#include "morph.hpp"
#include <iostream>
```

```cpp
int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray_crop = mm::_read_state("tmp/state/img_gray_crop_58.png");
// [pdi:state-io:end]

    mm::show(std::vector<mm::Image>{img_gray_crop, mm::prewitt(img_gray_crop)},
             MM_OUT,
             std::vector<std::string>{"Original", "Magnitude |grad f| (mm.prewitt)"},
             2);
    return 0;
}
```

```
Overwriting tmp/fig_03_prewitt.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_prewitt.cpp -o tmp/fig_03_prewitt \
  && ./tmp/fig_03_prewitt \
  && test -f "tmp/fig_03_prewitt.png" \
  || echo "  mm::show não gravou tmp/fig_03_prewitt.png"
```

```
[1] Original
[2] Magnitude |grad f| (mm.prewitt)
```

```
try:
    mm.show(mm.read("tmp/fig_03_prewitt.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_prewitt.png
    (ver a versao Python)")
```

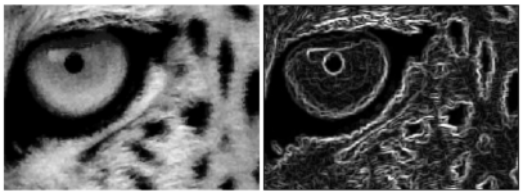


Figure 3.19: Operador de Prewitt: magnitude do gradiente, comparável ao Sobel mas sem a ponderação central. mm::prewitt devolve $|f|$ com clip.

3.6.4 *Unsharp Masking* (USM)

Unsharp Masking is a classic sharpening technique originating from analog photography, now widely used in image editing software. The central idea is to extract the **high-frequency components** of the image (edges and details) and add them back to the original with a weight k:

Table 3.3: Steps of *Unsharp Masking*.

Step	Operation	Description
1	$\bar{f} = f * G_\sigma$	Smooths with a Gaussian — retains low frequencies
2	$m = f - \bar{f}$	Mask: difference = high frequencies (edges)
3	$g = f + k \cdot m$	Weighted sum of the mask added to the original

Substituting step 2 into step 3 yields the compact expression:

$$g = f + k\,(f - f * G_\sigma) = (1 + k)\,f - k\,(f * G_\sigma) \tag{3.23}$$

The parameter k controls the intensity of the enhancement:

- $k = 0$: no enhancement ($g = f$);
- $k = 1$: classic USM — doubles the contribution of high frequencies;
- $k > 1$: *High Boost Filtering* — amplification beyond double, useful for highly blurred images.

> **Noise Amplification**
>
> USM does not distinguish edges from noise — both are high-frequency components. For high k, the noise present in the image is amplified along with the edges. Therefore, it is advisable to apply slight smoothing before USM on noisy images, or to use a small σ in the Gaussian.

To illustrate the steps of USM, Figure 3.20 applies the method to a 30×30 *patch* of the leopard image, using $\sigma = 1$ and $k = 1$. Initially, the image is smoothed by a Gaussian filter. Then, the high-frequency mask is obtained by the difference between the original and the smoothed image. Finally, this mask is added to the original image, enhancing edges and details. The figure presents the three steps of the process and the final enhancement result.

```
%%writefile tmp/fig_03_usm2.cpp
#define MM_OUT "tmp/fig_03_usm2.png"
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray_crop = mm::_read_state("tmp/state/img_gray_crop_58.png");
// [pdi:state-io:end]

    //| label: fig-03-usm2
    //| fig-cap: "Realce por *Unsharp Masking* num *patch* do leopardo: mm::usm faz suavização
    Gaussiana, subtrai da original (máscara de alta frequência) e reintroduz a máscara
    realçada."
    //| echo: true
    //| output: true

    mm::Image patch = mm::crop(img_gray_crop, 35, 65, 45, 75);

    mm::Image p_suave = mm::gaussian(patch, 7, 1.0);   // suavização Gaussiana
    mm::Image p_usm   = mm::usm(patch, 1.0);           // realce completo (k = 1.0)

    mm::show(std::vector<mm::Image>{patch, p_suave, p_usm},
            MM_OUT,
            std::vector<std::string>{"Patch original", "Suavizado ( =1)", "Realçado USM
(k=1)"}, 3);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(patch, "tmp/fig_03_usm2_0.png");
mm::write(p_suave, "tmp/fig_03_usm2_1.png");
mm::write(p_usm, "tmp/fig_03_usm2_2.png");
// [pdi:panel-io:end]
```

```
return 0;
}
```

```
Overwriting tmp/fig_03_usm2.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_usm2.cpp -o tmp/fig_03_usm2 \
  && ./tmp/fig_03_usm2 \
  && test -f "tmp/fig_03_usm2.png" \
  || echo "  mm::show não gravou tmp/fig_03_usm2.png"
```

```
[1] Patch original
[2] Suavizado ( =1)
[3] Realçado USM (k=1)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_03_usm2_0.png"),
            mm.read("tmp/fig_03_usm2_1.png"),
            mm.read("tmp/fig_03_usm2_2.png"),
        ],
        titles=[
            'Patch original',
            'Suavizado ( =1)',
            'Realçado USM (k=1)',
        ],
        cols=3,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_usm2_0.png (ver
    a versao Python)")
```

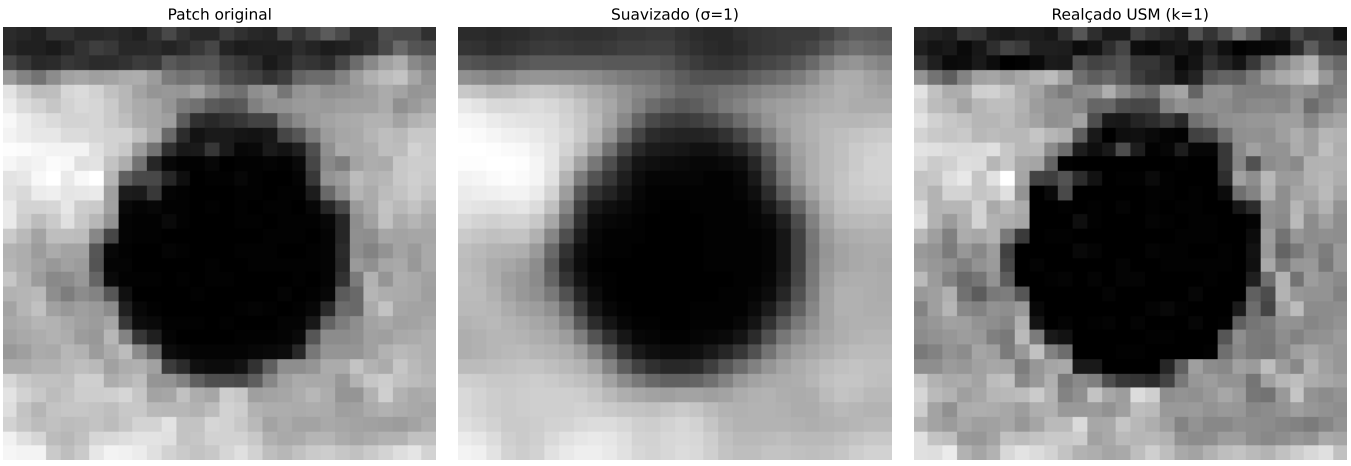


Figure 3.20: Realce por *Unsharp Masking* num *patch* do leopardo: mm::usm faz suavização Gaussiana, subtrai da original (máscara de alta frequência) e reintroduz a máscara realçada.

Figure 3.21 applies the USM method to a larger crop of the leopard image using $\sigma = 1$ and different values of the gain factor k. In all cases, the high-frequency mask is obtained by the difference between the original image and its version smoothed by a Gaussian filter. The parameter k controls the intensity of the enhancement: smaller values produce a subtle increase in sharpness, while larger values progressively reinforce edges and details. It is observed that, for high values of k, halos appear around edges and the noise present in the image becomes amplified.

```
%%writefile tmp/fig_03_usm.cpp
#define MM_OUT "tmp/fig_03_usm.png"
//| label: fig-03-usm
//| fig-cap: "*Unsharp Masking* on the leopard image with  =1 and k from 0.5 to 8.0. For k>2
halos appear at edges and background noise appears."
//| echo: true
//| output: true

#include "morph.hpp"
#include <vector>
#include <string>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray_crop = mm::_read_state("tmp/state/img_gray_crop_58.png");
// [pdi:state-io:end]

    mm::show(
        std::vector<mm::Image>{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)},
        MM_OUT,
        std::vector<std::string>{"Original", "USM k=0.5", "USM k=1.0", "USM k=3.0", "USM
k=5.0", "USM k=8.0"},
        3
    );
    return 0;
}
```

```
Overwriting tmp/fig_03_usm.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_usm.cpp -o tmp/fig_03_usm \
  && ./tmp/fig_03_usm \
  && test -f "tmp/fig_03_usm.png" \
  || echo "  mm::show não gravou tmp/fig_03_usm.png"
```

```
[1] Original
[2] USM k=0.5
[3] USM k=1.0
[4] USM k=3.0
[5] USM k=5.0
[6] USM k=8.0
```

```
try:
    mm.show(mm.read("tmp/fig_03_usm.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_usm.png (ver a
    versao Python)")
```

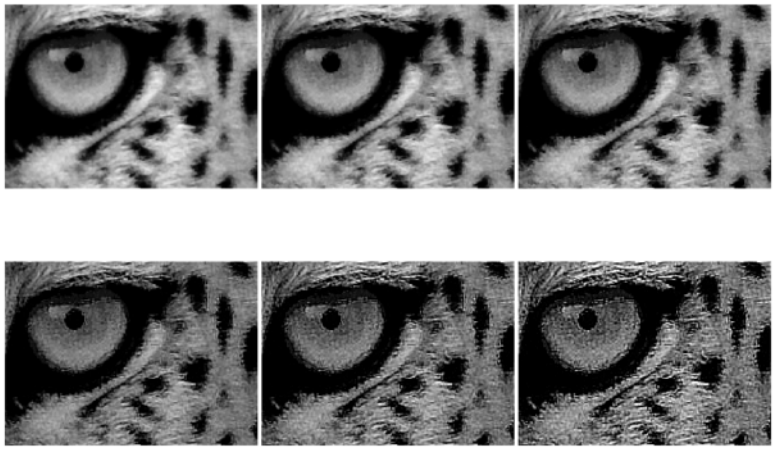


Figure 3.21: *Unsharp Masking* na imagem do leopardo com =1 e k de 0.5 a 8.0. Para k>2 surgem halos nas bordas e o ruído de fundo aparece.

3.6.5 Canny Detector

Canny combines four steps in sequence — Gaussian smoothing, Sobel gradient, non-maximum suppression, and dual-threshold hysteresis — to produce **thin, binary, and connected** edges. Unlike Sobel and Prewitt, the result is not a continuous gradient map, but a mask where each pixel is either an edge or not.

The central parameter is the threshold pair (T_{low}, T_{high}). Pixels with gradient above T_{high} are certain edges; below T_{low}, they are discarded. The ambiguous pixels — between the two thresholds — are decided by **hysteresis**: they become edges if they are connected to a certain edge, and are discarded otherwise. This avoids both the loss of weak segments of real edges and the inclusion of isolated noise. A common heuristic is $T_{high} = 3 \times T_{low}$.

```
%%writefile tmp/fig_03_canny.cpp
#define MM_OUT "tmp/fig_03_canny.png"
//| label: fig-03-canny
//| fig-cap: "Detector de Canny com diferentes pares de limiar: limiares baixos capturam mais
bordas (inclusive ruído); limiares altos retêm apenas as bordas mais fortes."
//| echo: true
//| output: true

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray_crop = mm::_read_state("tmp/state/img_gray_crop_58.png");
// [pdi:state-io:end]

    mm::show(
        std::vector<mm::Image>{
            img_gray_crop,
            mm::canny(img_gray_crop, 30, 90),
            mm::canny(img_gray_crop, 60, 180),
            mm::canny(img_gray_crop, 120, 240)
        },
        MM_OUT,
```

```
        std::vector<std::string>{"Original", "Canny (30/90)", "Canny (60/180)", "Canny
(120/240)"},
        4
    );
    return 0;
}
```

```
Overwriting tmp/fig_03_canny.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_canny.cpp -o tmp/fig_03_canny \
  && ./tmp/fig_03_canny \
  && test -f "tmp/fig_03_canny.png" \
  || echo "  mm::show não gravou tmp/fig_03_canny.png"
```

```
[1] Original
[2] Canny (30/90)
[3] Canny (60/180)
[4] Canny (120/240)
```

```
try:
    mm.show(mm.read("tmp/fig_03_canny.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_canny.png (ver
    a versao Python)")
```

Figure 3.22: Detector de Canny com diferentes pares de limiar: limiares baixos capturam mais bordas (inclusive ruído); limiares altos retêm apenas as bordas mais fortes.

Threshold Selection

A common heuristic is $T_{high} = 3 \times T_{low}$. Typical values depend on the image's gradient range — `cv2.Canny` accepts absolute values in $[0, 255]$. For images with variable contrast, computing the thresholds from the percentiles of the Sobel magnitude is more robust than using fixed values.

3.7 Order Filters: Median Filter

Order-statistic filters replace the central pixel with the value of a **percentile** of the neighborhood intensity distribution—unlike linear filters, which compute weighted combinations. The most important one is the **median filter**.

3.7.1 Impulsive Noise: Salt and Pepper

Salt-and-pepper noise replaces random pixels with extreme values: 0 (pepper, black) or 255 (salt, white). It is common in image transmission with bit errors and in cameras with defective sensors.

To understand why linear filters fail, consider a 3×3 neighborhood where a single pixel has been corrupted to 255:

$$\text{neighborhood} = \begin{bmatrix} 102 & 98 & 105 \\ 100 & \mathbf{255} & 97 \\ 103 & 99 & 101 \end{bmatrix}$$

Table 3.4: Mean vs. median with one corrupted pixel. The median ignores the outlier; the mean is shifted by ~40 levels.

Method	Calculation	Result
Mean	(102+98+…+255+…+101)/9	140
Median	{97,98,99,100,**101**,102,103,105,255}	101

⚠ Why do mean filters fail with impulsive noise?

The mean is sensitive to ***outliers***—a single pixel with a value of 255 in a neighborhood with values 100 raises the output to 140, spreading the noise throughout the image. The median, being a **robust estimator**, selects the central value of the sorted distribution, naturally discarding the extremes without any special adjustment.

The following example illustrates the behavior of the mean and the median in the presence of a pixel corrupted by impulsive noise. It is observed that the mean is strongly influenced by the extreme value (255), producing an estimate far from the predominant values of the neighborhood. Meanwhile, the median remains close to the original value of the region, evidencing its greater robustness to *outliers* and justifying its use in salt-and-pepper noise removal.

Figure 3.23 shows the effect of salt-and-pepper noise at different densities. The noise was generated by randomly replacing a fraction of pixels with minimum values (0, pepper) and maximum values (255, salt). As the density increases from 2% to 10%, the number of corrupted pixels grows, making the visual degradation more evident and hindering the perception of image details.

```
%%writefile tmp/fig_03_ruido.cpp
#define MM_OUT "tmp/fig_03_ruido.png"
//| label: fig-03-ruido
//| fig-cap: "Ruído sal e pimenta com densidades crescentes (2%, 5%, 10%): metade dos pixels
corrompidos vira sal (255), metade pimenta (0)."
//| echo: true
//| output: true

#include "morph.hpp"
#include <random>
#include <vector>
#include <string>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray_crop = mm::_read_state("tmp/state/img_gray_crop_58.png");
// [pdi:state-io:end]

    auto salt_pepper = [](const mm::Image& img, double prob) {
        mm::Image out = img;
        int h = out.h, w = out.w;
        std::mt19937 rng(42);
        std::uniform_int_distribution<int> dist_y(0, h - 1);
```

```
        std::uniform_int_distribution<int> dist_x(0, w - 1);
        std::uniform_real_distribution<double> dist_r(0.0, 1.0);
        int n = static_cast<int>(prob * h * w);
        for (int i = 0; i < n; ++i) {
            int y = dist_y(rng);
            int x = dist_x(rng);
            out.at(y, x) = (dist_r(rng) < 0.5) ? 0 : 255;
        }
        return out;
    };

    mm::Image n2  = salt_pepper(img_gray_crop, 0.02);
    mm::Image n5  = salt_pepper(img_gray_crop, 0.05);
    mm::Image n10 = salt_pepper(img_gray_crop, 0.10);

    mm::show({img_gray_crop, n2, n5, n10}, MM_OUT,
             {"Original", "Ruido 2%", "Ruido 5%", "Ruido 10%"}, 4);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray_crop, "tmp/fig_03_ruido_0.png");
mm::write(n2, "tmp/fig_03_ruido_1.png");
mm::write(n5, "tmp/fig_03_ruido_2.png");
mm::write(n10, "tmp/fig_03_ruido_3.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_03_ruido.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_ruido.cpp -o tmp/fig_03_ruido \
  && ./tmp/fig_03_ruido \
  && test -f "tmp/fig_03_ruido.png" \
  || echo "  mm::show não gravou tmp/fig_03_ruido.png"
```

```
[1] Original
[2] Ruido 2%
[3] Ruido 5%
[4] Ruido 10%
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_03_ruido_0.png"),
            mm.read("tmp/fig_03_ruido_1.png"),
            mm.read("tmp/fig_03_ruido_2.png"),
            mm.read("tmp/fig_03_ruido_3.png"),
        ],
        titles=[
            'Original',
            'Ruido 2%',
            'Ruido 5%',
            'Ruido 10%',
        ],
        cols=4,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_ruido_0.png
    (ver a versao Python)")
```

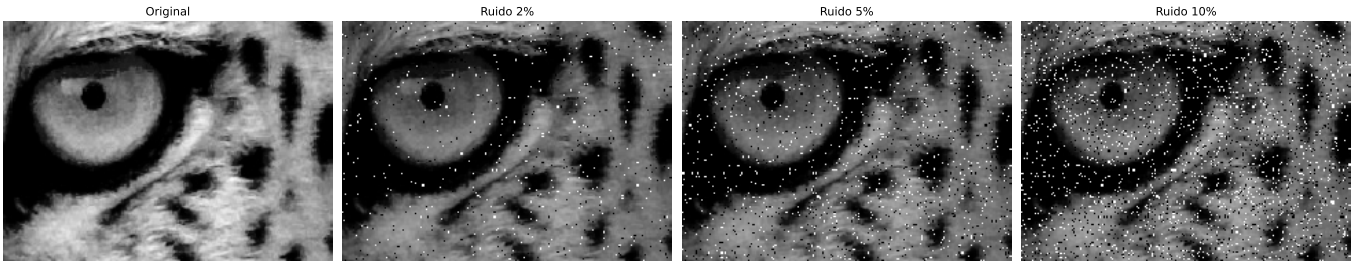


Figure 3.23: Ruído sal e pimenta com densidades crescentes (2%, 5%, 10%): metade dos pixels corrompidos vira sal (255), metade pimenta (0).

3.7.2 Median Filter

The median filter replaces each pixel with the **median value** of the pixels in its $n \times n$ neighborhood:

$$g(x, y) = \text{med}_{(s,t)\in\mathcal{V}_n}\{f(x + s, y + t)\} \tag{3.24}$$

The median value is the one occupying the central position when the n^2 values of the neighborhood are sorted. For a 3×3 window ($n^2 = 9$ pixels), the median is the 5th value in the sorted sequence.

To illustrate, consider the same 5×5 *patch* with a pixel artificially corrupted at $[1, 1]$:

The example confirms: even with the pixel corrupted to 255, the median returns the correct central value — the *outlier* occupies the last position in the ordering and is naturally discarded.

Because it is based on ordering rather than summation, the median has three fundamental properties that distinguish it from linear filters:

- **Robust** to impulsive noise — outliers go to the ends of the ordered sequence and do not affect the central value;
- **Edge-preserving** — abrupt intensity transitions are maintained, since the median selects a value that already exists in the neighborhood, without creating new intermediate levels;
- **Nonlinear** — it cannot be expressed as a convolution, therefore `mm::conv` does not apply; use `cv2.medianBlur`.

Figure 3.24 compares different salt-and-pepper noise removal techniques applied to an image with 10% corrupted pixels. The Gaussian, Mean, Median, Bilateral, and Morphological filters (next chapter) were evaluated, allowing observation of the trade-off between noise removal and detail preservation. In general, the mean and Gaussian filters reduce noise but tend to blur edges, while the median filter performs better for impulsive noise. The bilateral filter preserves edges better, and the morphological filter removes a good portion of corrupted pixels without excessively degrading the image structure.

```
%%writefile tmp/fig_03_ruido_filtros.cpp
#define MM_OUT "tmp/fig_03_ruido_filtros.png"
// Compile with: g++ -std=c++17 -O2 -o program program.cpp -lmorph
#include <random>
#include "morph.hpp"
#include <filesystem>

// salt and pepper noise generator
//| label: fig-03-ruido-filtros
//| fig-cap: "Filtros para ruído sal e pimenta (10%): Gaussiano, Média e Mediana. Bilateral e
morfológico (open+close) ficam só na trilha Python - bilateral não tem equivalente em
morph.hpp e morfologia é do próximo capítulo."
//| echo: true
//| output: true
```

```
mm::Image salt_pepper(const mm::Image& img, double prob) {
    mm::Image out = img;
    int h = out.h, w = out.w;
    std::mt19937 rng(42);
    std::uniform_int_distribution<int> dist_y(0, h - 1);
    std::uniform_int_distribution<int> dist_x(0, w - 1);
    std::uniform_real_distribution<double> dist_p(0.0, 1.0);
    for (int i = 0; i < static_cast<int>(prob * h * w); ++i) {
        int y = dist_y(rng);
        int x = dist_x(rng);
        out.at(y, x) = (dist_p(rng) < 0.5) ? 0 : 255;
    }
    return out;
}

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray_crop = mm::_read_state("tmp/state/img_gray_crop_58.png");
// [pdi:state-io:end]

    mm::Image noisy = salt_pepper(img_gray_crop, 0.10);

    mm::Image f_gauss   = mm::gaussian(noisy, 5, 1.0);
    mm::Image f_media   = mm::blur(noisy, 5);
    mm::Image f_median3 = mm::median(noisy, 3);
    mm::Image f_median5 = mm::median(noisy, 5);

    mm::show(
        std::vector<mm::Image>{img_gray_crop, noisy, f_gauss, f_media, f_median3, f_median5},
        MM_OUT,
        std::vector<std::string>{"Original", "Ruido 10%", "Gaussiano 5x5", "Media 5x5",
            "Mediana 3x3", "Mediana 5x5"},
        3
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray_crop, "tmp/fig_03_ruido_filtros_0.png");
mm::write(noisy, "tmp/fig_03_ruido_filtros_1.png");
mm::write(f_gauss, "tmp/fig_03_ruido_filtros_2.png");
mm::write(f_media, "tmp/fig_03_ruido_filtros_3.png");
mm::write(f_median3, "tmp/fig_03_ruido_filtros_4.png");
mm::write(f_median5, "tmp/fig_03_ruido_filtros_5.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_03_ruido_filtros.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_ruido_filtros.cpp -o tmp/fig_03_ruido_filtros \
  && ./tmp/fig_03_ruido_filtros \
  && test -f "tmp/fig_03_ruido_filtros.png" \
  || echo "  mm::show não gravou tmp/fig_03_ruido_filtros.png"
```

```
[1] Original
[2] Ruido 10%
[3] Gaussiano 5x5
[4] Media 5x5
[5] Mediana 3x3
```

```
[6] Mediana 5x5
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_03_ruido_filtros_0.png"),
            mm.read("tmp/fig_03_ruido_filtros_1.png"),
            mm.read("tmp/fig_03_ruido_filtros_2.png"),
            mm.read("tmp/fig_03_ruido_filtros_3.png"),
            mm.read("tmp/fig_03_ruido_filtros_4.png"),
            mm.read("tmp/fig_03_ruido_filtros_5.png"),
        ],
        titles=[
            'Original',
            'Ruido 10%',
            'Gaussiano 5x5',
            'Media 5x5',
            'Mediana 3x3',
            'Mediana 5x5',
        ],
        cols=3,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_03_ruido_filtros_0.png (ver a versao Python)")
```

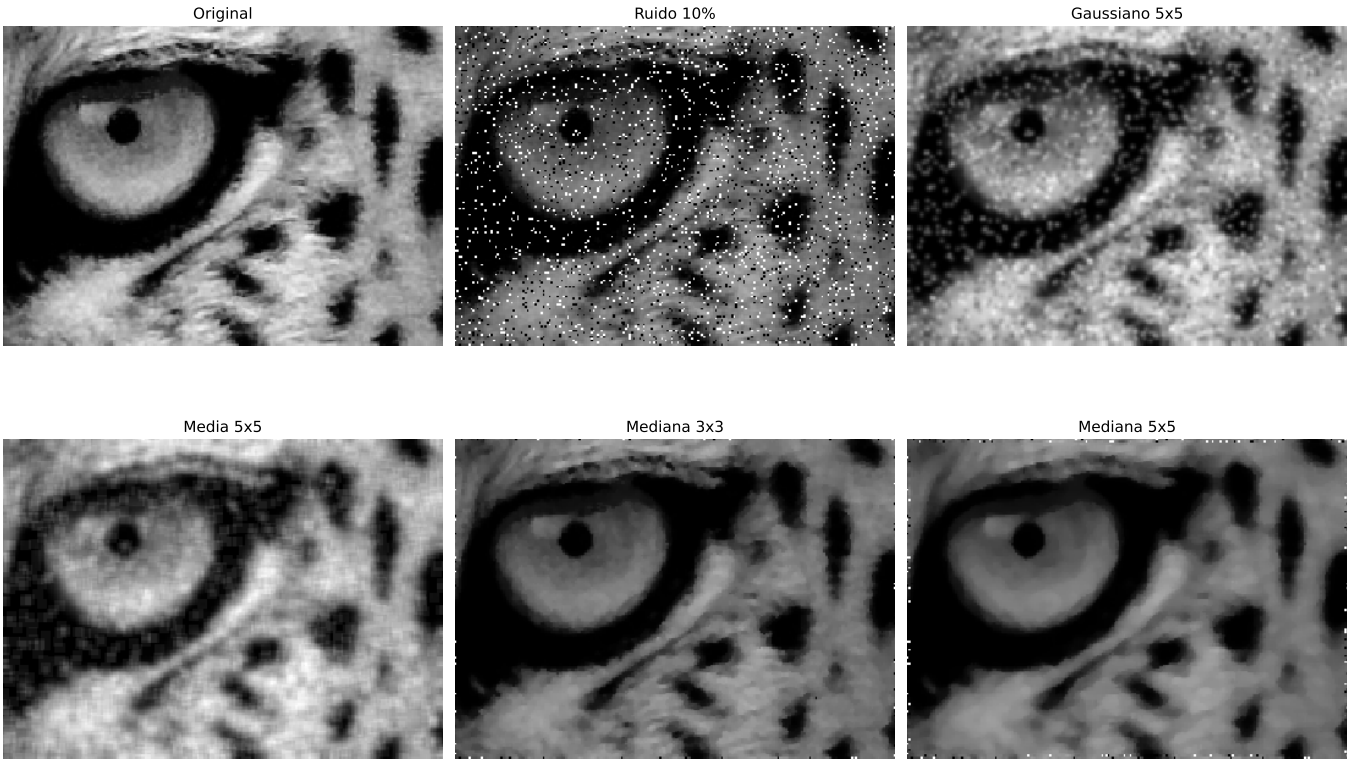


Figure 3.24: Filtros para ruído sal e pimenta (10%): Gaussiano, Média e Mediana. Bilateral e morfológico (open+close) ficam só na trilha Python — bilateral não tem equivalente em morph.hpp e morfologia é do próximo capítulo.

3.8 Practical Application: Preprocessing for Segmentation

In practice, the techniques in this chapter are rarely used in isolation. A typical **preprocessing pipeline** combines multiple steps in sequence, tailored to the image type and application. Figure 3.25 illustrates a

complete pipeline:

1. **Histogram equalization (CLAHE):** normalizes contrast regardless of lighting conditions;
2. **Gaussian filter:** smooths acquisition noise without destroying edges;
3. **Edge detection (Sobel/Canny):** extracts relevant structures for segmentation.

> **Order matters**
>
> The order of operations affects the final result. In general: **(1) intensity normalization → (2) noise reduction → (3) enhancement/segmentation**. Reversing the order can amplify noise or cause edges to be lost before detection.

```
%%writefile tmp/fig_03_pipeline.cpp
#define MM_OUT "tmp/fig_03_pipeline.png"
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray_crop = mm::_read_state("tmp/state/img_gray_crop_58.png");
// [pdi:state-io:end]

    // img_gray_crop is provided as an already-valid mm::Image variable

    //| label: fig-03-pipeline
    //| fig-cap: "*Pipeline* de pre-processing: equalization → Gaussian → Canny. The Python
    track uses CLAHE instead of global equalization; CLAHE has no equivalent in morph.hpp."
    //| echo: true
    //| output: true

    mm::Image img_eq    = mm::equalize(img_gray_crop);     // Step 1: global equalization
    mm::Image img_gauss = mm::gaussian(img_eq, 5, 0);      // Step 2: Gaussian
    mm::Image edges     = mm::canny(img_gauss, 50, 150);   // Step 3: Canny

    mm::Image edges_direct = mm::canny(img_gray_crop, 50, 150);   // Direct Canny, no
pre-processing

    mm::show(
        std::vector<mm::Image>{img_gray_crop, img_eq, img_gauss, edges, edges_direct},
        MM_OUT,
        std::vector<std::string>{"Original", "1. Equalized", "2. Gaussian", "3. Canny
(pipeline)", "Canny (direct)"},
        5
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray_crop, "tmp/fig_03_pipeline_0.png");
mm::write(img_eq, "tmp/fig_03_pipeline_1.png");
mm::write(img_gauss, "tmp/fig_03_pipeline_2.png");
mm::write(edges, "tmp/fig_03_pipeline_3.png");
mm::write(edges_direct, "tmp/fig_03_pipeline_4.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_03_pipeline.cpp
```

```
!g++ -I. -std=c++17 tmp/fig_03_pipeline.cpp -o tmp/fig_03_pipeline \
  && ./tmp/fig_03_pipeline \
  && test -f "tmp/fig_03_pipeline.png" \
  || echo "  mm::show não gravou tmp/fig_03_pipeline.png"
```

```
[1] Original
[2] 1. Equalized
[3] 2. Gaussian
[4] 3. Canny (pipeline)
[5] Canny (direct)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_03_pipeline_0.png"),
            mm.read("tmp/fig_03_pipeline_1.png"),
            mm.read("tmp/fig_03_pipeline_2.png"),
            mm.read("tmp/fig_03_pipeline_3.png"),
            mm.read("tmp/fig_03_pipeline_4.png"),
        ],
        titles=[
            'Original',
            '1. Equalizado',
            '2. Gaussiano',
            '3. Canny (pipeline)',
            'Canny (direto)',
        ],
        cols=5,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_03_pipeline_0.png
    (ver a versao Python)")
```

Figure 3.25: *Pipeline* de pré-processamento: equalização → Gaussiano → Canny. A trilha Python usa CLAHE no lugar da equalização global; CLAHE não tem equivalente em morph.hpp.

3.9 Summary

This chapter presented the main spatial domain processing techniques, from direct pixel manipulation to neighborhood filtering:

- **Point operations:** saturated arithmetic (`mm::addm`, `mm::subm`) and bitwise logic (`mm::band`, `mm::bor`, `mm::bnot`) for ROI cropping and image combination; *alpha blending* (`mm::blend`) for weighted fusion with weight $\alpha \in [0, 1]$.
- **Histogram:** discrete function of intensity distribution; visualized with `mm::histImg` and computed with `mm::hist`; the basis for tonal diagnosis and for equalization and specification techniques.
- **Equalization:** automatic redistribution of intensities via the CDF (`mm::equalize`), with the adaptive CLAHE variant for local contrast control.

- **Histogram specification:** transfer of the tonal profile from a reference image via inverse CDF mapping — a generalization of equalization to arbitrary distributions.
- **Correlation and convolution:** sliding window mechanism implemented in `mm::conv` (`cv2.filter2D`); distinguished by the 180° rotation of the *kernel* — relevant only for asymmetric kernels.
- **Smoothing filters:** mean (uniform *kernel*, blurs edges proportionally to size) and Gaussian (radial weighting, separable, no *ringing*, preserves edges better).
- **Sharpening filters:** Laplacian (w_4/w_8, isotropic second derivative), Sobel (first-order directional gradient, with magnitude $|\nabla f|$ and direction θ) and *Unsharp Masking* (high-frequency amplification with parameter k).
- **Median filter:** nonlinear, robust to outliers, edge-preserving — superior to linear filters for salt-and-pepper noise.
- **Practical pipeline:** CLAHE → Gaussian → Canny chaining as a preprocessing strategy; `mm::drawImgKernel` for didactic visualization of the sliding window.

Chapter 4 will address **mathematical morphology** (erosion, dilation, opening, and closing), exploring in depth the `mm::ero` and `mm::dil` functions from the `morph.py` library. Next, Chapter 5 will present **frequency domain processing**, focusing on the Fourier Transform and spectral filtering techniques.

3.10 🤖 Using Gemini Notebook as a Complementary Tutor

In this edition, we encourage the use of **Gemini Notebook** as a complementary learning tool. This AI tool uses exclusively the documents provided by the author as its knowledge base, ensuring responses consistent with the book's content—including the functions of the `morph.py` library and the experiments conducted in this chapter.

For each chapter, we have prepared a specific project on the platform containing the chapter's PDF, the *notebooks*, and auxiliary materials. We suggest exploring in particular:

- **Study Guide:** a structured summary of the concepts, ideal for review before exams;
- **Chat:** ask questions about equalization, convolution, filters, and pipelines directly with the tutor;
- **Frequently Asked Questions:** typical questions about the difference between mean and median, USM, Laplacian vs. Sobel.

🎓 Study with the Intelligent Tutor

To interact with the content of this chapter, access the following *link*. The environment contains teaching materials in different formats, generated from the chapter's **PDF**. On the platform, explore especially the **Study Guide** and **Chat** options to deepen your understanding.
🚀 ACCESS GEMINI NOTEBOOK: CHAPTER 03

🌐 Language and Programming Language

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

⚠️ Notice on AI-Generated Content

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

3.11 Exercise List

1. **(10%)** Explain the difference between **convolution** and **cross-correlation**. For which types of *kernels* are the results identical? Give an example of an asymmetric *kernel* (such as Sobel G_x) and show numerically that the results differ by applying it to the 5×5 patch from the chapter in both ways.

2. **(15%)** Consider a 5×5 image with intensities concentrated between levels 3 and 5 (low contrast, 3 bits). Manually apply the equalization algorithm from Table 3.1, filling in all the columns of the table (k, $h[k]$, $p[k]$, cdf$[k]$, lut$[k]$). Verify the result with `mm::equalize`.

3. **(15%)** Using `mm::conv`, apply the mean filter with kernels of size 3×3, 9×9, and 21×21 to the mandrill image. For each version, compute the **PSNR** (*Peak Signal-to-Noise Ratio*) relative to the original:

$$\text{PSNR} = 10\log_{10}\left(\frac{255^2}{\text{MSE}}\right), \quad \text{MSE} = \frac{1}{MN}\sum_{i,j}(f-g)^2$$

   Plot the PSNR as a function of kernel size and explain what the progressive decline indicates about the relationship between smoothing and information loss.

4. **(15%)** Using `add_salt_pepper` with a density of 5%, apply and compare: (a) `mm::conv` with a 3×3 mean filter, (b) `cv2.GaussianBlur` with $\sigma = 1$, (c) `cv2.medianBlur` with a 3×3 window, and (d) `cv2.medianBlur` with a 5×5 window. Display the images with `mm::show` in a 2×4 grid (row 1: images, row 2: histograms via `mm::histImg`). Explain why the median outperforms linear filters using the argument from Table 3.4.

5. **(15%)** Implement `mm::conv0` using only vectorized NumPy operations — without Python loops and without `cv2.filter2D` — using the *stride tricks* operator (`np.lib.stride_tricks.sliding_window_view`). Compare the result and execution time with `mm::conv0` (loops) and `mm::conv` (cv2) for 3×3 and 15×15 kernels on the mandrill image.

6. **(15%)** Apply *Unsharp Masking* with $\sigma = 1$ and $k \in \{0.5, 1.0, 2.0, 4.0\}$ using the `usm` function from the chapter. For each value of k: (a) compute the absolute difference $|g - f|$, (b) display the images and the differences with `mm::show`, and (c) plot the histogram of the differences with `mm::histImg`. Identify from which k onward the artifacts (halos and noise amplification) become visually unacceptable.

7. **(15%)** Choose a publicly available X-ray or tomography image (e.g., via `mm::read` from a URL) and design a preprocessing pipeline with at least 4 sequential steps, justifying each choice based on the concepts from the chapter. Display with `mm::show` in a grid: the original image, each intermediate stage, and the final result along with their histograms (`mm::histImg`).

Chapter References

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

- Gonzalez (2018) for the concepts of intensity operations, histogram, convolution, and spatial filtering.
- Szeliski (2022) for computer vision and practical applications of filtering.
- Bradski (2008) for the practical implementation with OpenCV and `morph.py`.

Figure 3.18 presents a small notebook suggestion with the possible structure to be adopted by the class. The suggestion in this section may help you in completing your digital notebook, which is part of the continuous assessment (CA) of this course — see Section 3.9.

The assessment activities proposed here may be carried out in groups of up to 3 (three) students, with the respective completion deadline.

To work on the **Google Colab** environment, you must have a Google account (Gmail). The suggestions in this notebook do not follow any pre-established standard, but are just coding examples that can assist in testing the proposed digital image processing (DIP) and computer vision (CV) tasks.

Note: The codes below are just examples for testing the main functions of the task.

The proposed exercise list (EPs, in Portuguese) contains the following experiments:

- **EP01** – Calculate the negative of the image `mario.png`.
- **EP02** – Swap the color channels of the image `mario.png`, transforming it from RGB to GRB.
- **EP03** – Represent the image `mario.png` in the HSV color space.
- **EP04** – Threshold the image `mario.png` in a binary fashion, using the HSV color space.
- **EP05** – Swap the color channels of the image `mario.png`, transforming it from RGB to BGR.
- **EP06** – Using the `img1.pgm` and `img2.pgm` images, present: (a) the result of `img1` AND `img2`; and (b) the result of `img1` OR `img2`.
- **EP07** – Provide the histogram and the negative of the image `pout.tif`, making the necessary adjustments so that the image appears with good visual quality.

3.12 💻 Practical Part with Programming Exercises

🎯 Objective of this Notebook

The notebook allows you to develop, validate, organize, and test solutions for **Programming Exercises (EPs)** in interactive environments, such as Colab, using the same test cases as Moodle, and only copy them there when recording the official grade.

Download

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

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # C++ track build artifacts (.cpp, binary, PNGs)

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")

# The kernel is Python even in the C++ track: `mm` (morph.py) is used by the
# simulators, by the display of the figures the C++ binary generates and by the
# mm::Image state between cells. cpp=True also downloads the compiled track
# (morph.hpp + stb_image*.h), used in the #include of the %%writefile *.cpp cells.
import config
config.setup(testsuite=True, cpp=True)
from morph import mm
from testsuite import TestSuite
```

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

Running the Tests

To evaluate the tests, run `TestSuite("EP03_01.extension").run()` in a new cell, replacing the extension with the one corresponding to the language used (`.py`, `.java`, `.c`, `.cpp`, `.js`, or `.r`). The system downloads the test cases from GitHub, runs the program, and calculates the grade automatically.

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

```
code = """
from morph import mm
# ... your code here ...
"""
TestSuite("EP03_01").run_code(code)
```

3.12.1 EP03_01 ➕ Saturated Addition of a Constant

In video surveillance systems, cameras in environments with variable lighting produce underexposed images. Adjusting brightness by **saturated addition of a constant** is the simplest operation for immediate correction, being applied in real time on embedded camera *chips* and in preprocessing *pipelines* of mobile robots.

See Figure 3.26 for a simulation of this EP.

3.12.1.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Constant:** Read the integer k (value to be added).
3. **Data:** Read the integer values of the original matrix row by row.
4. **Mapping:** For each pixel p, compute the new value using the equation:

$$p' = \text{clip}(p + k)$$

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

3.12.1.2 📌 Computational Constraints

- **Saturation (*Clipping*):** Values must be confined to the interval $[0, 255]$:

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

- **Type:** The final result must be an integer (no decimal places).
- k **can be negative:** negative values darken the image; positive values lighten it.

3.12.1.3 🧠 Theoretical Background

Parameter	Type	Visual Impact
$k > 0$	Integer	Lightens the image; pixels near 255 saturate to white
$k < 0$	Integer	Darkens the image; pixels near 0 saturate to black
$k = 0$	Integer	Image unchanged

3.12.1.4 📦 Input and Output Specification (VPL)

Input:

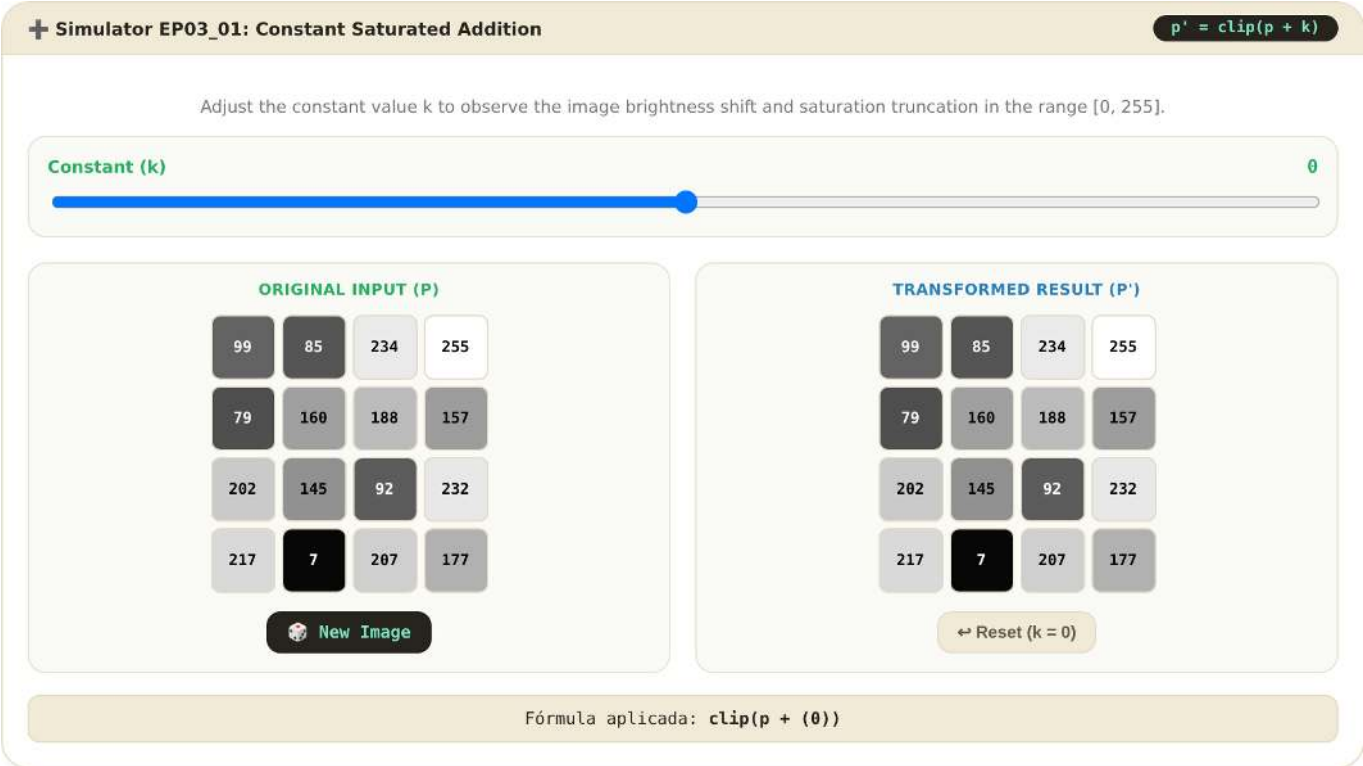
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer k.
- Following lines: Integer elements of the original matrix.

Output:

- Transformed matrix with L rows and C columns, integer values separated by spaces.

3.12.1.5 📌 Examples

Input	Output	Observation
2 3 50 0 100 200 210 240 255	50 150 250 255 255 255	Saturation at 255 for high pixels
1 4 -30 0 20 200 255	0 0 170 225	Saturation at 0 for low pixels



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap03/sim-ep0301-adicao.html

Figure 3.26: Simulador EP03_01: Saturated Addition of Constant (p' = clip(p + k))

```
%%writefile EP03_01.cpp
// your solution
```

```
Overwriting EP03_01.cpp
```

```
TestSuite("EP03_01.cpp").run()
```

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

3.12.2 EP03_02 🔀 Alpha Blending of Two Images

In nuclear medicine, images from different modalities (computed tomography and magnetic resonance imaging) are fused to aid in diagnosis. **Weighted blending** (*alpha blending*) is the fundamental operation of this process, allowing the radiologist to interactively control the weight of each modality in the displayed image.

See Figure 3.27 for a simulation of this EP.

3.12.2.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Parameter:** Read the real value $\alpha \in [0, 1]$.
3. **Data:** Read the integer values of matrix f_1 (image 1) followed by those of matrix f_2 (image 2).
4. **Mapping:** For each position (i, j), compute:

$$g(i, j) = \text{clip}\left(\text{round}\left(\alpha \cdot f_1(i, j) + (1 - \alpha) \cdot f_2(i, j)\right)\right)$$

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

3.12.2.2 📌 Computational Constraints

- **Rounding:** Apply `round` before converting to integer.
- **Saturation:** Constrain to the interval $[0, 255]$ with $\text{clip}(x) = \max(0, \min(255, x))$.
- **Float operation:** Perform the operation in floating point before rounding.

3.12.2.3 🧠 Theoretical Background

Value of α	Result
$\alpha = 1.0$	Only f_1
$\alpha = 0.5$	Arithmetic mean of f_1 and f_2
$\alpha = 0.0$	Only f_2

3.12.2.4 📦 Input and Output Specification (VPL)

Input:

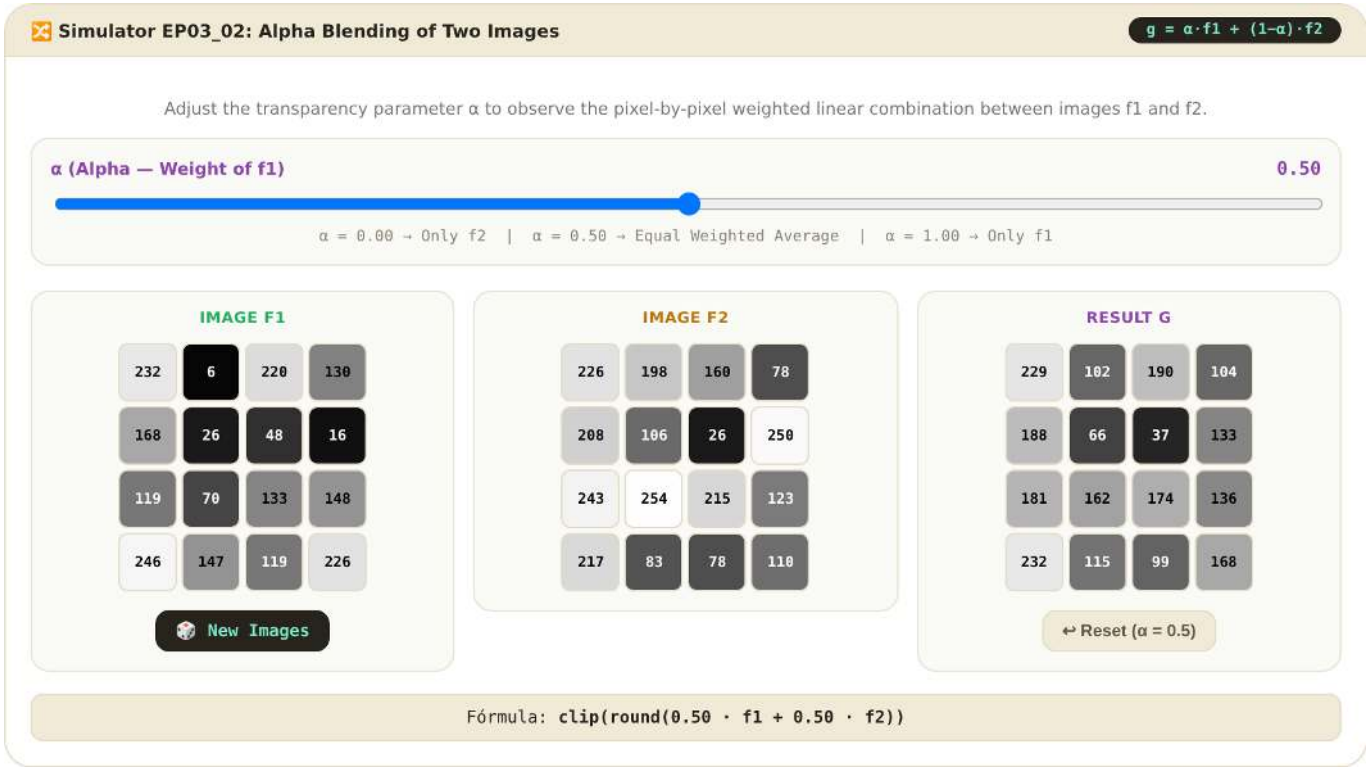
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Real α.
- Following lines: Elements of f_1 (L lines with C values each).
- Following lines: Elements of f_2 (L lines with C values each).

Output:

- Resulting $L \times C$ matrix.

3.12.2.5 📌 Examples

Input	Output	Observation
1 3 0.5 0 100 200 100 200 50	50 150 125	Mean between the two images
1 3 1.0 10 20 30 90 80 70	10 20 30	Only f_1 (alpha=1)



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap03/sim-ep0302-blending.html

Figure 3.27: Simulator EP03_02: Alpha Blending of Two Images (g = · f1 + (1−) · f2)

```
%%writefile EP03_02.cpp
// your solution
```

```
Overwriting EP03_02.cpp
```

```
TestSuite("EP03_02.cpp").run()
```

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

3.12.3 EP03_03 🎭 Image Inversion (Photographic Negative)

In radiology, X-ray images are traditionally viewed as negatives: bones appear in black on a white background. The **photographic negative** operation is routinely applied in PACS (*Picture Archiving and Communication Systems*) to facilitate the detection of fractures and bone densities.

See Figure 3.28 for a simulation of this EP.

3.12.3.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Data:** Read the integer values of the original matrix.
3. **Mapping:** For each pixel p, compute the negative:

$$p' = 255 - p$$

4. **Output:** Display the resulting matrix $L \times C$.

3.12.3.2 📌 Computational Constraints

- **No clipping required:** The result of $255 - p$ with $p \in [0, 255]$ is always $\in [0, 255]$.

- **Integer type:** The output must consist of integer values.
- **Logical equivalence:** The operation is identical to the bitwise `NOT` (`mm::bnot`) on 8-bit images.

3.12.3.3 🧠 Theoretical Foundation

Original Pixel p	Negative Pixel p'	Observation
0 (black)	255 (white)	Total inversion
128 (medium gray)	127 (medium gray)	Central value
255 (white)	0 (black)	Total inversion

3.12.3.4 📦 Input and Output Specification (VPL)

Input:

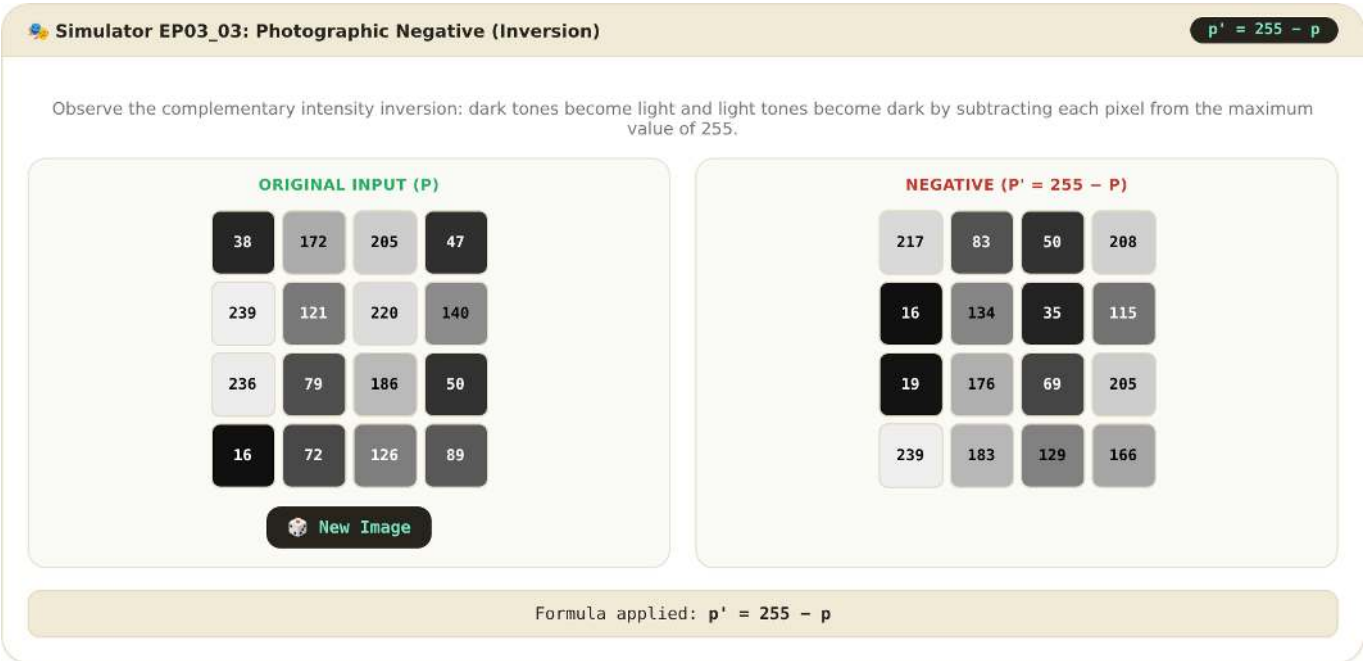
- Line 1: Integer L.
- Line 2: Integer C.
- Following lines: Integer elements of the original matrix.

Output:

- Negative matrix with L rows and C columns.

3.12.3.5 📌 Examples

Input	Output	Observation
140 128 200 255	255 127 55 0	Inversion of each pixel
2 2 10 20 30 40	245 235225 215	Inverted 2x2 matrix



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap03/sim-ep0303-inversao.html

Figure 3.28: EP03_03 Simulator: Image Inversion — Photographic Negative (p' = 255 − p)

```
%%writefile EP03_03.cpp
// your solution
```

```
Overwriting EP03_03.cpp
```

```
TestSuite("EP03_03.cpp").run()
```

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

3.12.4 EP03_04 📊 Histogram Equalization (L bits)

In remote sensing satellite images, variation in illumination throughout the day produces low-contrast images. **Histogram equalization** is automatically applied in satellites such as Landsat to redistribute tones, revealing details of vegetation, relief, and urban areas that are invisible in the original image.

See Figure 3.29 for a simulation of this EP.

3.12.4.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows), C (columns), and B (number of bits, with $L_{\max} = 2^B$).
2. **Data:** Read the pixel matrix f with values in $[0, 2^B - 1]$.
3. **Histogram:** Compute $h[k]$ = number of pixels with intensity k, for $k = 0 \ldots 2^B - 1$.
4. **Probability:** $p[k] = h[k]/(L \cdot C)$.
5. **CDF:** $\text{cdf}[k] = \sum_{j=0}^{k} p[j]$; cumulative distribution function.
6. **LUT:** $\text{lut}[k] = \text{round}\,(\text{cdf}[k] \cdot (2^B - 1))$; *Look-Up Table.*
7. **Application:** $g[i, j] = \text{lut}[f[i, j]]$.
8. **Output:** Display the equalized matrix of size $L \times C$.

3.12.4.2 📌 Computational Constraints

- **Rounding:** Use mathematical rounding (`round`) in the LUT.
- **Bits:** The number of levels is 2^B (e.g., $B = 3 \Rightarrow 8$ levels, $B = 8 \Rightarrow 256$ levels).
- **Cumulative CDF:** $\text{cdf}[k] = \sum_{j=0}^{k} p[j]$, with $\text{cdf}[2^B - 1] = 1.0$.

3.12.4.3 🧠 Theoretical Foundation

Step	Operation	Formula
1	Histogram	$h[k] \leftarrow$ number of pixels with intensity k
2	Probability	$p[k] = h[k]/(L \cdot C)$
3	CDF	$\text{cdf}[k] = \sum_{j=0}^{k} p[j]$
4	LUT	$\text{lut}[k] = \text{round}(\text{cdf}[k] \cdot (2^B - 1))$
5	Application	$g[i, j] = \text{lut}[f[i, j]]$

3.12.4.4 📦 Input and Output Specification (VPL)

Input:

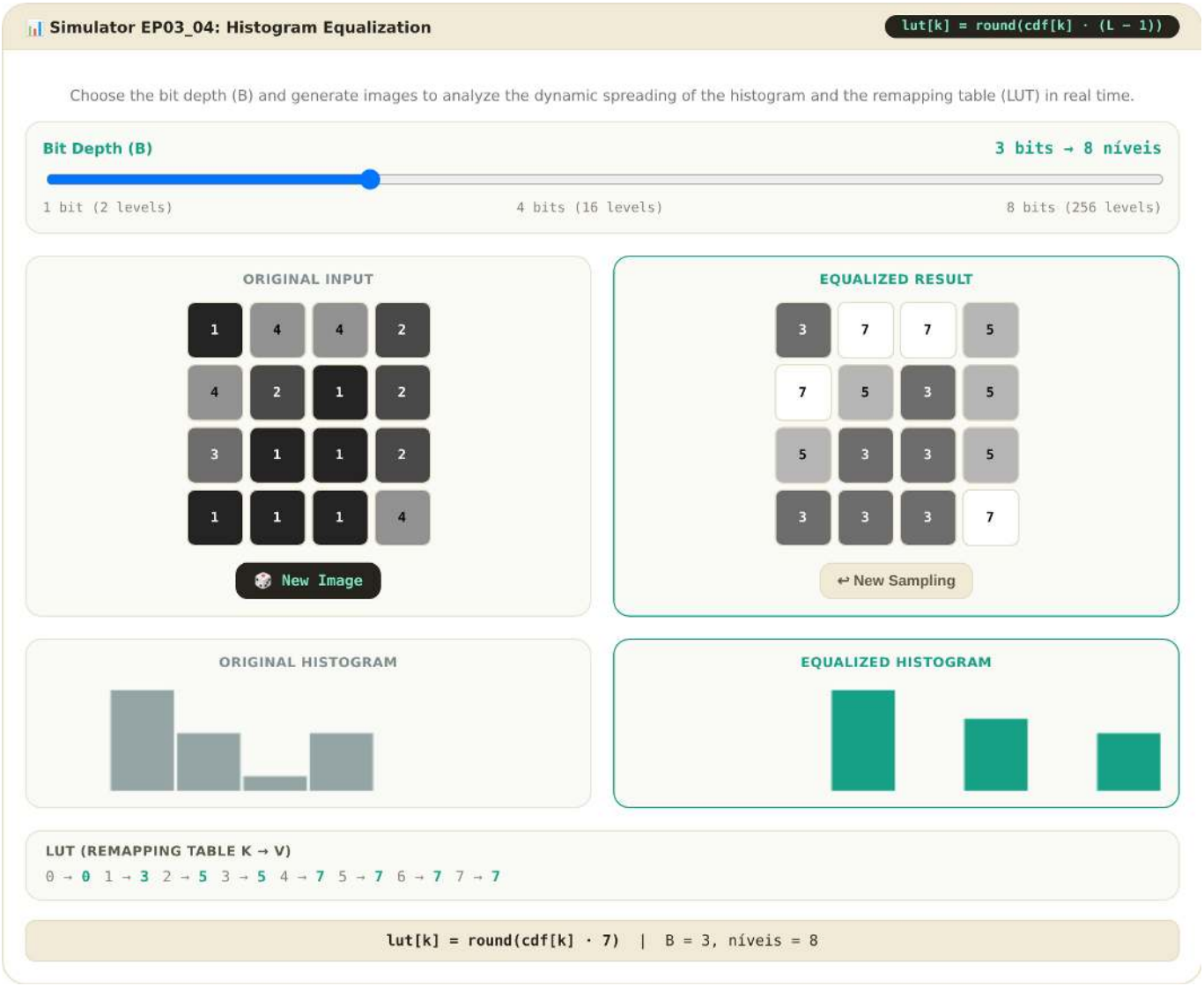
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer B (number of bits).
- Following lines: Integer elements of the matrix.

Output:

- Equalized matrix with L rows and C columns.

3.12.4.5 📌 Examples

Input	Output	Observation
5 5 3 3 4 2 3 4 4 3 3 4 3 2 3 4 3 2 3 4 3 2 3 4 3 2 3 4	5 7 1 5 7 7 5 5 7 5 1 5 7 5 1 5 7 5 1 5 7 5 1 5 7	Example with 3 bits from the chapter
1 4 3 0 0 7 7	0 0 7 7	Extreme bimodal histogram



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap03/sim-ep0304-equalizacao.html

Figure 3.29: Simulator EP03_04: Histogram Equalization (Levels L = 2^B)

```
%%writefile EP03_04.cpp
// your solution
```

```
Overwriting EP03_04.cpp
```

```
TestSuite("EP03_04.cpp").run()
```

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

3.12.5 EP03_05 🔲 Binary AND Mask Application

In industrial computer vision inspection systems, it is necessary to isolate regions of interest (ROI) in part images to verify manufacturing defects. The **bitwise AND operation with a binary mask** is the fundamental mechanism for precisely cropping the inspection area, zeroing all pixels outside it.

See Figure 3.30 for a simulation of this exercise.

3.12.5.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Data:** Read the pixel matrix f (values $\in [0, 255]$).
3. **Mask:** Read the binary matrix m (values: only 0 or 255).
4. **Mapping:** For each pixel (i, j), apply the bitwise AND:

$$g(i, j) = f(i, j) \text{ AND } m(i, j)$$

where $255 =$ `11111111` and $0 =$ `00000000` in binary.

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

3.12.5.2 📌 Computational Constraints

- **AND with 255:** p AND $255 = p$ (all bits preserved).
- **AND with 0:** p AND $0 = 0$ (all bits zeroed).
- **Mask:** The only possible values in the mask are 0 and 255.
- **Implementation:** In Python, bitwise AND between integers uses the `&` operator.

3.12.5.3 🧠 Theoretical Foundation

Pixel f	Mask m	Result f AND m
any v	255 (`11111111`)	v (preserved)
any v	0 (`00000000`)	0 (zeroed)

3.12.5.4 📦 Input and Output Specification (VPL)

Input:

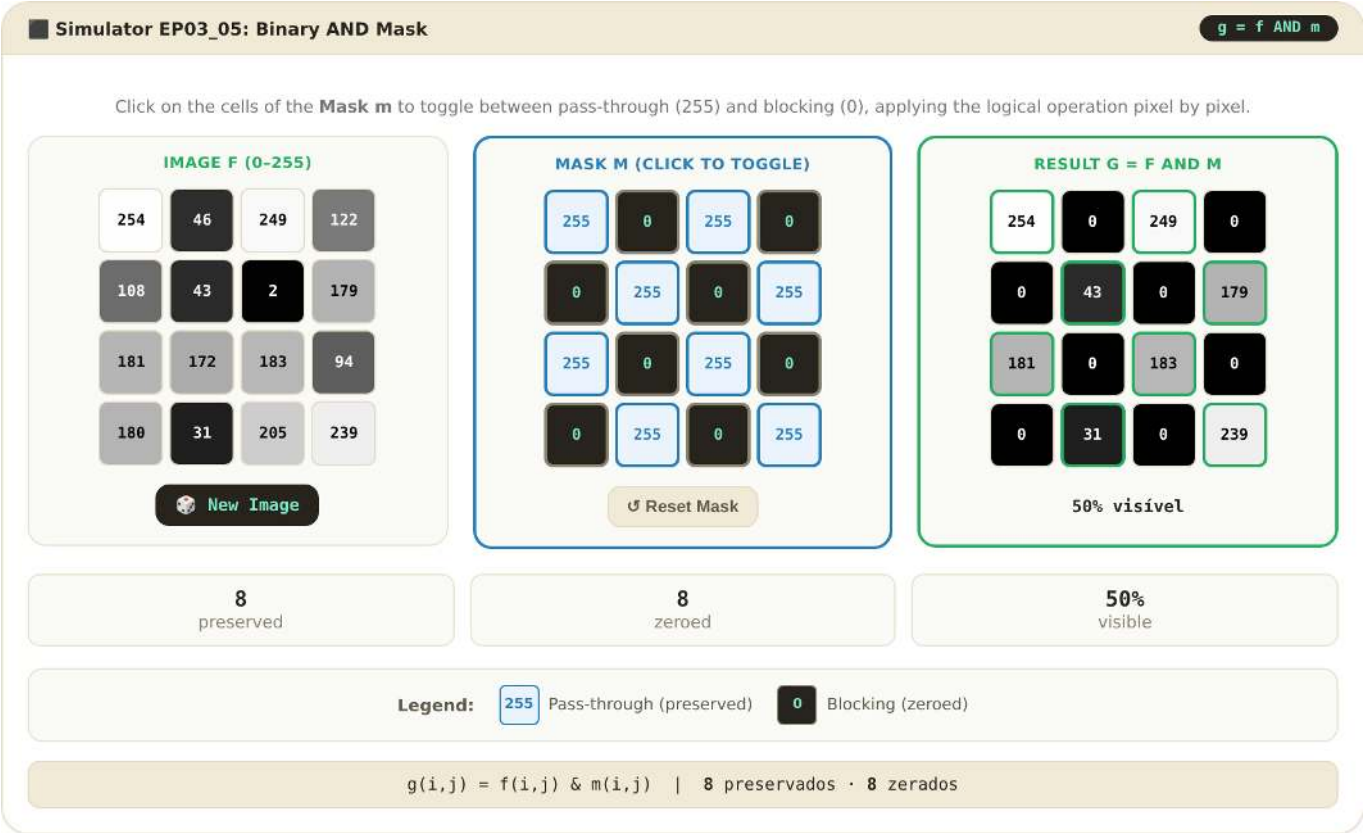
- Line 1: Integer L.
- Line 2: Integer C.
- Following lines: Elements of f (L rows).
- Following lines: Elements of m (L rows with values 0 or 255).

Output:

- Resulting $L \times C$ matrix.

3.12.5.5 📌 Examples

Input	Output	Observation
2 3 100 150 200 50 80 120 255 255 0 0 255 255	100 150 0 0 80 120	Mask selects region
1 4 10 20 30 40 255 0 255 0	10 0 30 0	Alternating preserved/zeroed



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap03/sim-ep0305-mascara.html

Figure 3.30: EP03_05 Simulator: Binary AND Mask Application

```
%%writefile EP03_05.cpp
// your solution
```

```
Overwriting EP03_05.cpp
```

```
TestSuite("EP03_05.cpp").run()
```

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

3.12.6 EP03_06 N×N Mean Filter *Kernel*

In autonomous vehicle cameras, images captured under rain or fog exhibit Gaussian noise. The **mean filter** is widely used for real-time noise reduction, being implemented directly in the **ISP** (*Image Signal Processor*) of **CMOS** (*Complementary Metal-Oxide-Semiconductor*) sensors.

CMOS sensors are the image sensors used in most modern cameras (*smartphones*, *webcams*, automotive cameras, etc.). They convert light into electrical signals, and the ISP processes these signals in real time — applying operations such as noise reduction, white balance, and other image adjustments.

See the simulation of this EP in Figure 3.31.

3.12.6.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows), C (columns), and N (kernel size, always odd).
2. **Data:** Read the pixel matrix f.
3. **Mean Filter:** For each **internal** pixel (i, j) (without borders), compute:

$$g(i,j) = \text{round}\left(\frac{1}{N^2}\sum_{s=-(r)}^{r}\sum_{t=-(r)}^{r} f(i+s,\, j+t)\right), \quad r = \lfloor N/2 \rfloor$$

4. **Border Handling:** Pixels at the border (where the $N \times N$ window exceeds the limits) must be **copied directly** from the original without modification.
5. **Output:** Display the resulting $L \times C$ matrix.

3.12.6.2 📌 Computational Constraints

- **Radius:** $r = \lfloor N/2 \rfloor$ (half of the kernel, integer).
- **Internal pixels:** (i, j) with $r \leq i < L - r$ and $r \leq j < C - r$.
- **Rounding:** Use mathematical rounding before converting to integer.
- **No clipping:** The average of values $\in [0, 255]$ remains in $[0, 255]$.

3.12.6.3 🧠 Theoretical Background

Size N	Coefficient	Pixels in the window	Effect
3	$1/9 \approx 0.111$	9	Smooth
5	$1/25 = 0.04$	25	Medium
7	$1/49 \approx 0.020$	49	Strong

3.12.6.4 📦 Input and Output Specification (VPL)

Input:

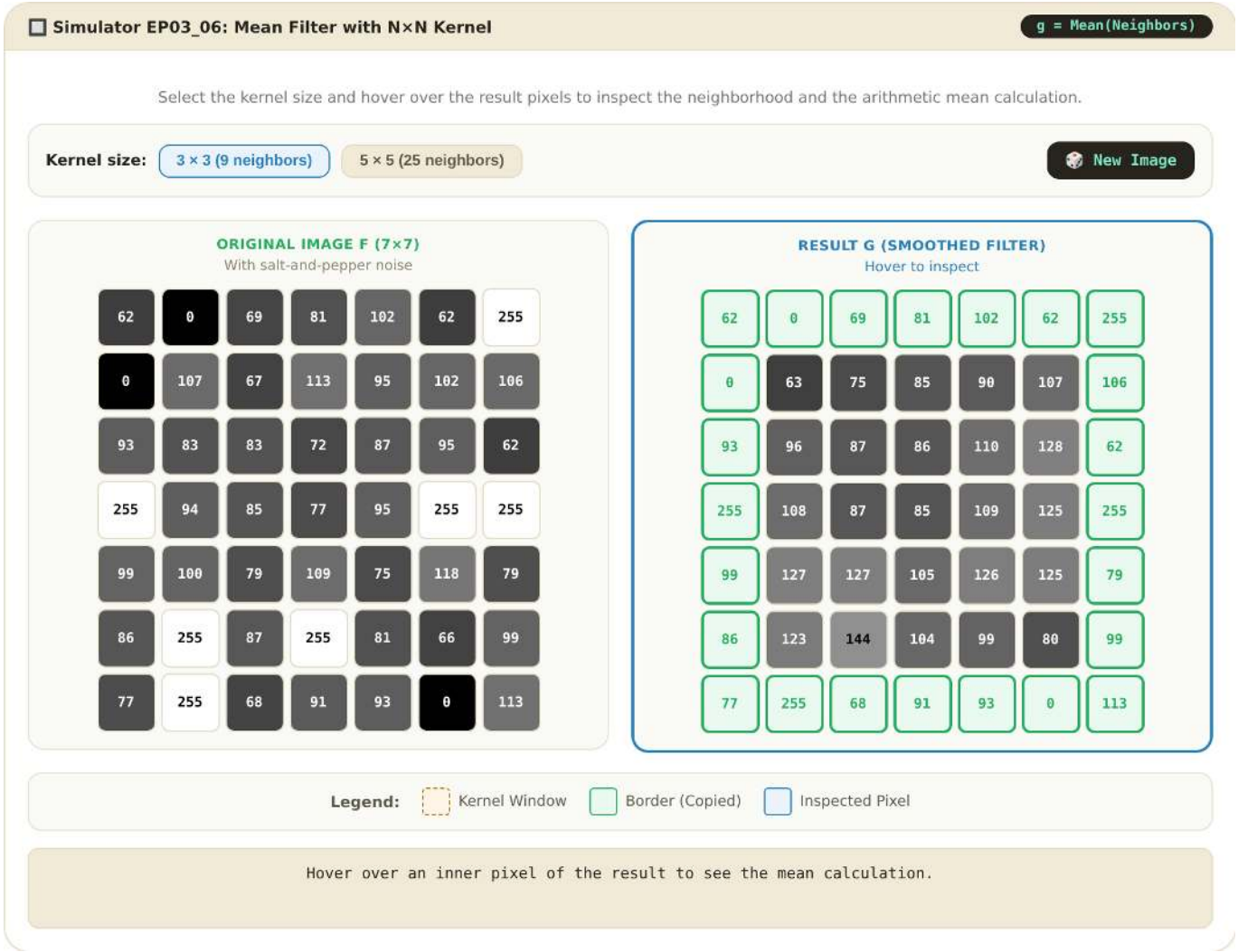
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer N (odd, $N \geq 3$).
- Following lines: Elements of the original matrix.

Output:

- Filtered $L \times C$ matrix.

3.12.6.5 📌 Examples

Input	Output	Observation
3 3 3 10 20 30 40 50 60 70 80 90	10 20 30 40 50 60 70 80 90	Only border (3×3 = all border)
5 5 3 0 0 0 0 0 0 0 0 0 0 0 0 100 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 11 11 11 0 0 11 11 11 0 0 11 11 11 0 0 0 0 0 0	Isolated pixel: all 9 internal pixels whose 3×3 window includes the value 100 receive round(100/9)=11



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap03/sim-ep0306-media.html

Figure 3.31: Simulator EP03_06: Average Filter with N×N Kernel

```
%%writefile EP03_06.cpp
// your solution
```

```
Overwriting EP03_06.cpp
```

```
TestSuite("EP03_06.cpp").run()
```

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

3.12.7 EP03_07 🔍 Laplacian Operator (w4) for Edge Enhancement

In high-resolution tomography, the sharpness of edges between tissues is critical for diagnosis. The **Laplacian operator** is widely used in medical image preprocessing *pipelines* to automatically enhance anatomical contours before segmentation, avoiding manual intervention by the radiologist.

See Figure 3.32 for a simulation of this EP.

3.12.7.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Data:** Read the pixel matrix f.
3. **Laplacian (w4):** For each **internal** pixel (i, j) with $1 \leq i < L-1$, $1 \leq j < C-1$, compute:

$$\nabla^2 f(i,j) = f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1) - 4 \cdot f(i,j)$$

4. **Enhancement:** Compute the enhanced image:

$$g(i,j) = \text{clip}(f(i,j) - \nabla^2 f(i,j))$$

5. **Border:** Border pixels are copied directly: $g(i,j) = f(i,j)$.
6. **Output:** Display the enhanced matrix $L \times C$.

3.12.7.2 📌 Computational Constraints

- ***Kernel* w4:** $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ — only 4-neighbors.
- **Saturation:** $\text{clip}(x) = \max(0, \min(255, x))$ applied to the enhancement result.
- **No rounding:** The Laplacian uses only integer additions/subtractions.

3.12.7.3 🧠 Theoretical Background

Region	$\nabla^2 f$	Enhancement Effect
Uniform	≈ 0	No change
Rising edge	< 0	Pixel lightened
Falling edge	> 0	Pixel darkened

3.12.7.4 📦 Input and Output Specification (VPL)

Input:

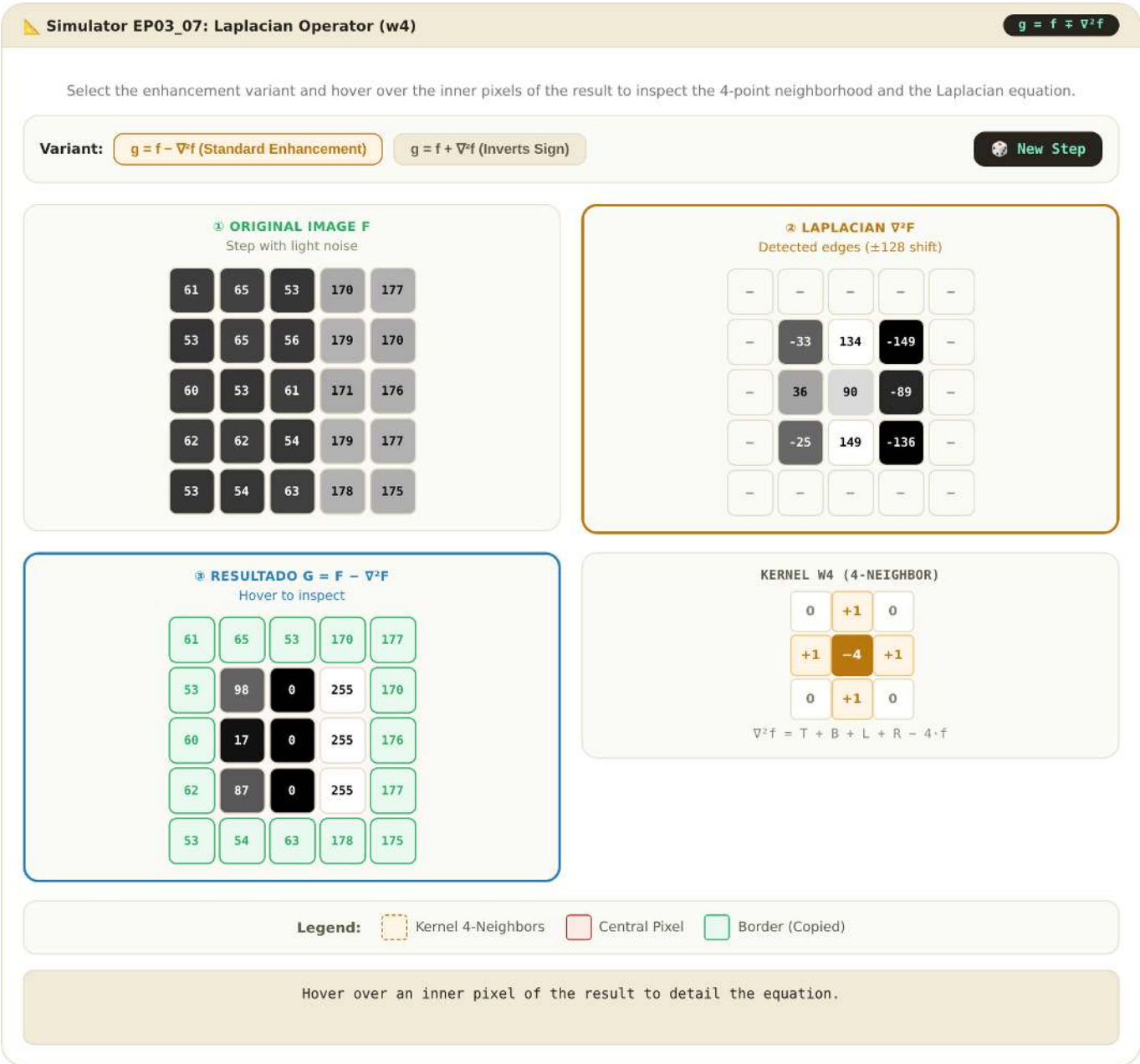
- Line 1: Integer L.
- Line 2: Integer C.
- Following lines: Elements of the original matrix.

Output:

- Enhanced matrix $L \times C$.

3.12.7.5 📌 Examples

Input	Output	Observation
3 3 0 0 0 0 100 0 0 0 0	0 0 0 0 255 0 0 0 0	Isolated peak: lap=−400, g=100−(−400)=500 → clip=255
3 3 50 50 50 50 50 50 50 50 50	50 50 50 50 50 50 50 50 50	Uniform region: Laplacian=0, no change



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap03/sim-ep0307-laplaciano.html

Figure 3.32: Simulator EP03_07: Laplacian Operator (w4) for Edge Enhancement

```
%%writefile EP03_07.cpp
// your solution
```

```
Overwriting EP03_07.cpp
```

```
TestSuite("EP03_07.cpp").run()
```

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

3.12.8 EP03_08 🧭 Sobel Gradient: Gx and Gy

In Mars exploration rovers (such as Perseverance), obstacle detection is performed in real time by stereoscopic cameras. The **Sobel operator** computes the directional gradient of the scene and is used in the edge detection algorithm to identify rocks, cracks, and terrain unevenness that could compromise navigation.

See Figure 3.33 for a simulation of this EP.

3.12.8.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Data:** Read the matrix f.
3. **Gx and Gy:** For each **internal** pixel (i, j) with $1 \leq i < L-1$, $1 \leq j < C-1$:

$$G_x(i,j) = [f(i-1,j+1) + 2f(i,j+1) + f(i+1,j+1)] - [f(i-1,j-1) + 2f(i,j-1) + f(i+1,j-1)]$$

$$G_y(i,j) = [f(i+1,j-1) + 2f(i+1,j) + f(i+1,j+1)] - [f(i-1,j-1) + 2f(i-1,j) + f(i-1,j+1)]$$

4. **Magnitude:** $|\nabla f(i,j)| = \text{clip}(\text{round}(\sqrt{G_x^2 + G_y^2}))$.
5. **Edge:** Edge pixels receive magnitude 0.
6. **Output:** Display the $L \times C$ magnitude.

3.12.8.2 📌 Computational Constraints

- **Rounding:** Apply `round` before converting to integer.
- **Saturation:** $\text{clip}(x) = \max(0, \min(255, x))$.
- **Square root:** Use $\sqrt{G_x^2 + G_y^2}$ (not the approximation $|G_x| + |G_y|$).

3.12.8.3 🧠 Theoretical Foundation

Operator	Detects	Diagonal coefficients
G_x	Vertical edges	± 1
G_y	Horizontal edges	± 1
$\lvert\nabla f\rvert$	All edges	Combined

3.12.8.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.

- Following lines: Matrix elements.

Output:

- Gradient magnitude, $L \times C$ matrix.

3.12.8.5 📌 Examples

Input	Output	Observation
3 3 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0	Null image: zero gradient
3 3 0 0 255 0 0 255 0 0 255	0 0 0 0 255 0 0 0 0	Central vertical edge: high Gx

```
%%writefile EP03_08.cpp
// your solution
```

```
Overwriting EP03_08.cpp
```

```
TestSuite("EP03_08.cpp").run()
```

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

3.12.9 EP03_09 📡 3×3 Median Filter

Synthetic aperture radar (SAR) images used in environmental and military monitoring suffer from a specific type of noise called *speckle*, which has characteristics similar to salt-and-pepper noise. The **median filter** is the standard method for removing this noise because it preserves the edges of structures while eliminating spurious points.

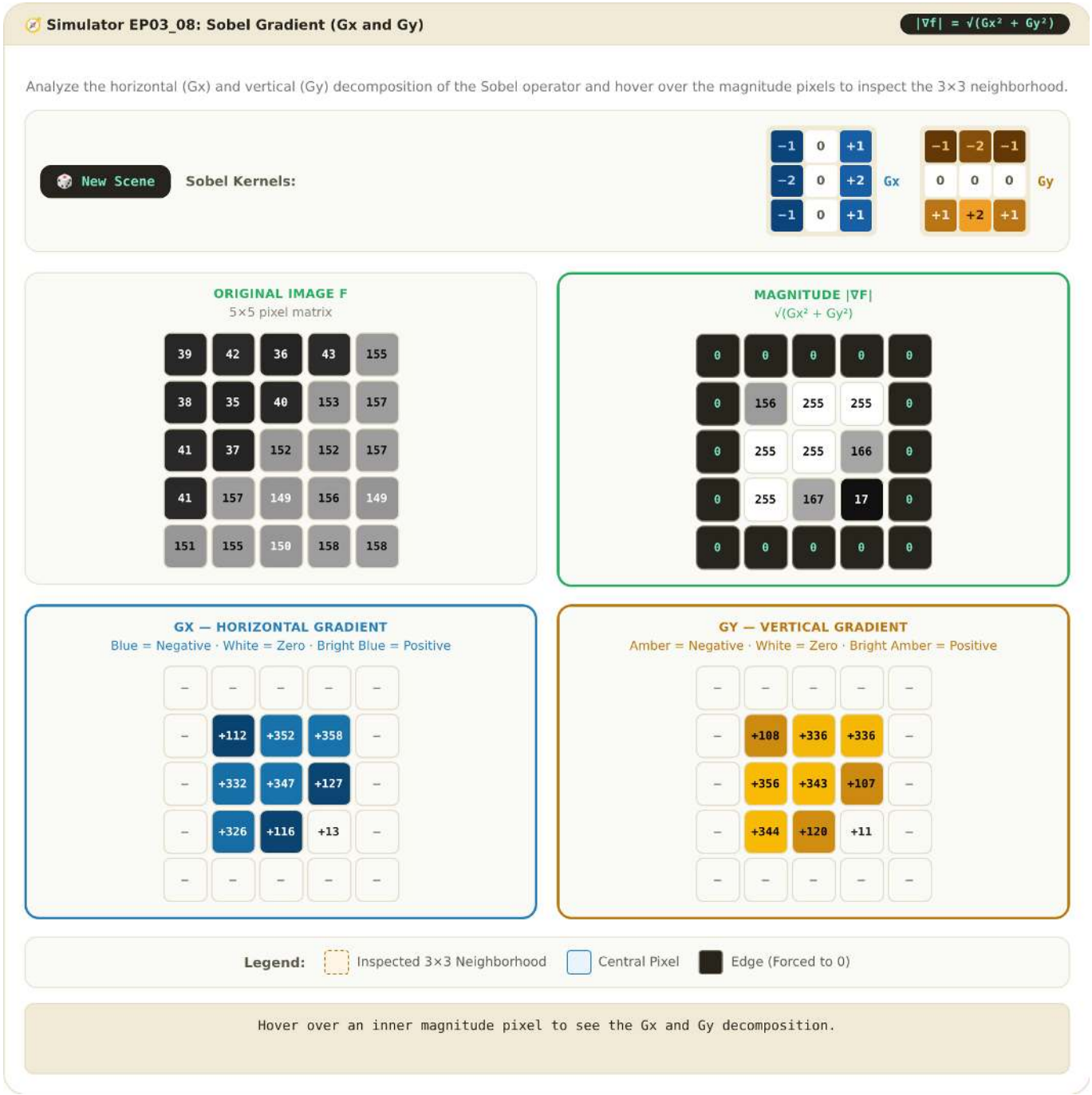
See Figure 3.34 for a simulation of this exercise.

3.12.9.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Data:** Read the pixel matrix f.
3. **3×3 Median Filter:** For each **internal** pixel (i, j) with $1 \leq i < L-1$, $1 \leq j < C-1$:
   - Collect the 9 pixels from the 3×3 neighborhood: $\{f(i+s, j+t) : s, t \in \{-1, 0, 1\}\}$.
   - Sort the 9 values in ascending order.
   - Assign $g(i, j)$ to the central value (index position 4, considering index 0).

$$g(i,j) = \text{median}\{f(i+s, j+t) : s, t \in \{-1, 0, 1\}\}$$

4. **Border:** Copy directly: $g(i, j) = f(i, j)$.
5. **Output:** Display the filtered $L \times C$ matrix.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap03/sim-ep0308-sobel.html

Figure 3.33: EP03_08 Simulator: Sobel Gradient (Gx and Gy)

3.12.9.2 📌 Computational Constraints

- **Window:** Always $3 \times 3 = 9$ elements.
- **Median:** The central element of the sorted sequence (index 4 from 0 to 8).
- **No clipping:** The median of values in $[0, 255]$ remains in $[0, 255]$.
- **Nonlinear:** The median filter cannot be expressed as a linear convolution.

3.12.9.3 🧠 Theoretical Background

Noise	Mean Filter	Median Filter
Salt and pepper (0 or 255)	Spreads the noise	Removes without distorting edges
Gaussian	Reduces effectively	Reduces partially

3.12.9.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Following lines: Matrix elements.

Output:

- Filtered $L \times C$ matrix.

3.12.9.5 📌 Examples

Input	Output	Observation
3 3 100 100 100 100 0 100 100 100 100	100 100 100 100 100 100 100 100 100	Black point removed: median of 8×100+1×0 = 100
3 3 50 50 50 50 255 50 50 50 50	50 50 50 50 50 50 50 50 50	White point (salt) removed

```
%%writefile EP03_09.cpp
// your solution
```

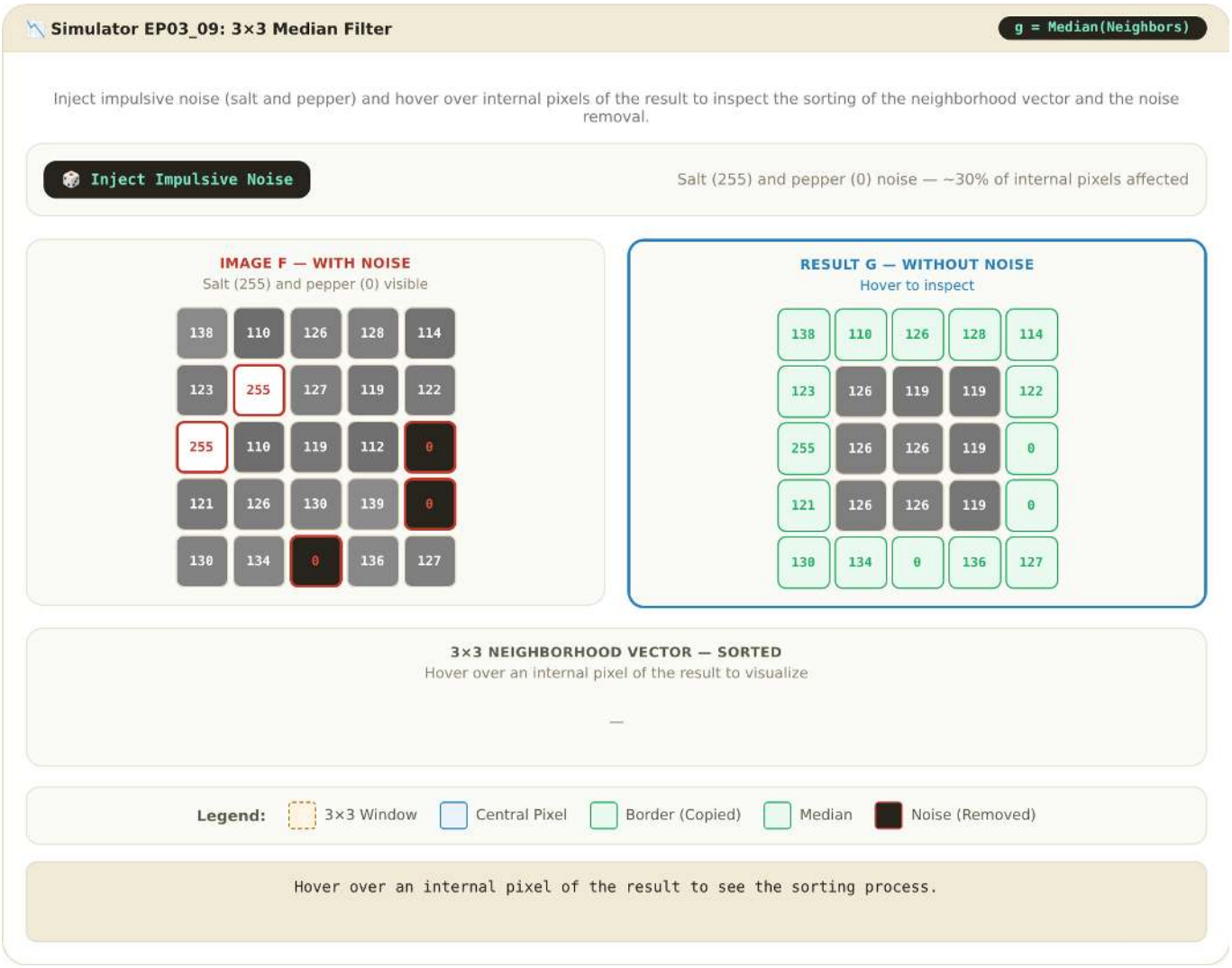
```
Overwriting EP03_09.cpp
```

```
TestSuite("EP03_09.cpp").run()
```

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

3.12.10 EP03_10 ✨ *Unsharp Masking* (USM)

In digitization systems for historical documents and works of art, image sharpness is essential for reading handwritten texts and ornamental details. ***Unsharp Masking*** **(USM)** is the standard sharpening



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap03/sim-ep0309-mediana.html

Figure 3.34: EP03_09 Simulator: 3×3 Median Filter

enhancement algorithm used in professional scanners and software such as Adobe Photoshop, controlled by the parameter k that determines the enhancement intensity.

See Figure 3.35 for a simulation of this EP.

3.12.10.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Parameter:** Read the real value k (enhancement intensity, $k \geq 0$).
3. **Data:** Read the pixel matrix f.
4. **Smoothing:** Compute $\bar{f}$ with a 3×3 averaging filter (internal pixels only; borders preserved):

$$\bar{f}(i,j) = \frac{1}{9} \sum_{s=-1}^{1} \sum_{t=-1}^{1} f(i+s, j+t)$$

5. **High-frequency mask:** $m(i,j) = f(i,j) - \bar{f}(i,j)$.
6. **USM Enhancement:** For each internal pixel:

$$g(i,j) = \text{clip}\left(\text{round}\left(f(i,j) + k \cdot m(i,j)\right)\right)$$

7. **Border:** $g(i,j) = f(i,j)$ (direct copy).
8. **Output:** Display the enhanced matrix $L \times C$.

3.12.10.2 📌 Computational Constraints

- **Rounding:** Apply `round` before clipping.
- **Saturation:** $\text{clip}(x) = \max(0, \min(255, x))$.
- **Float operations:** Compute $\bar{f}$ and m in floating point before rounding the final result.
- $k = 0$**:** No enhancement — the output is identical to the input (except for borders).

3.12.10.3 🧠 Theoretical Foundation

Step	Operation	Description
1	$\bar{f} = f * \frac{1}{9}\mathbf{1}_{3\times3}$	Smoothing (low frequencies)
2	$m = f - \bar{f}$	Mask (high frequencies)
3	$g = \text{clip}(\text{round}(f + k \cdot m))$	Weighted enhancement

3.12.10.4 📦 Input and Output Specification (VPL)

Input:

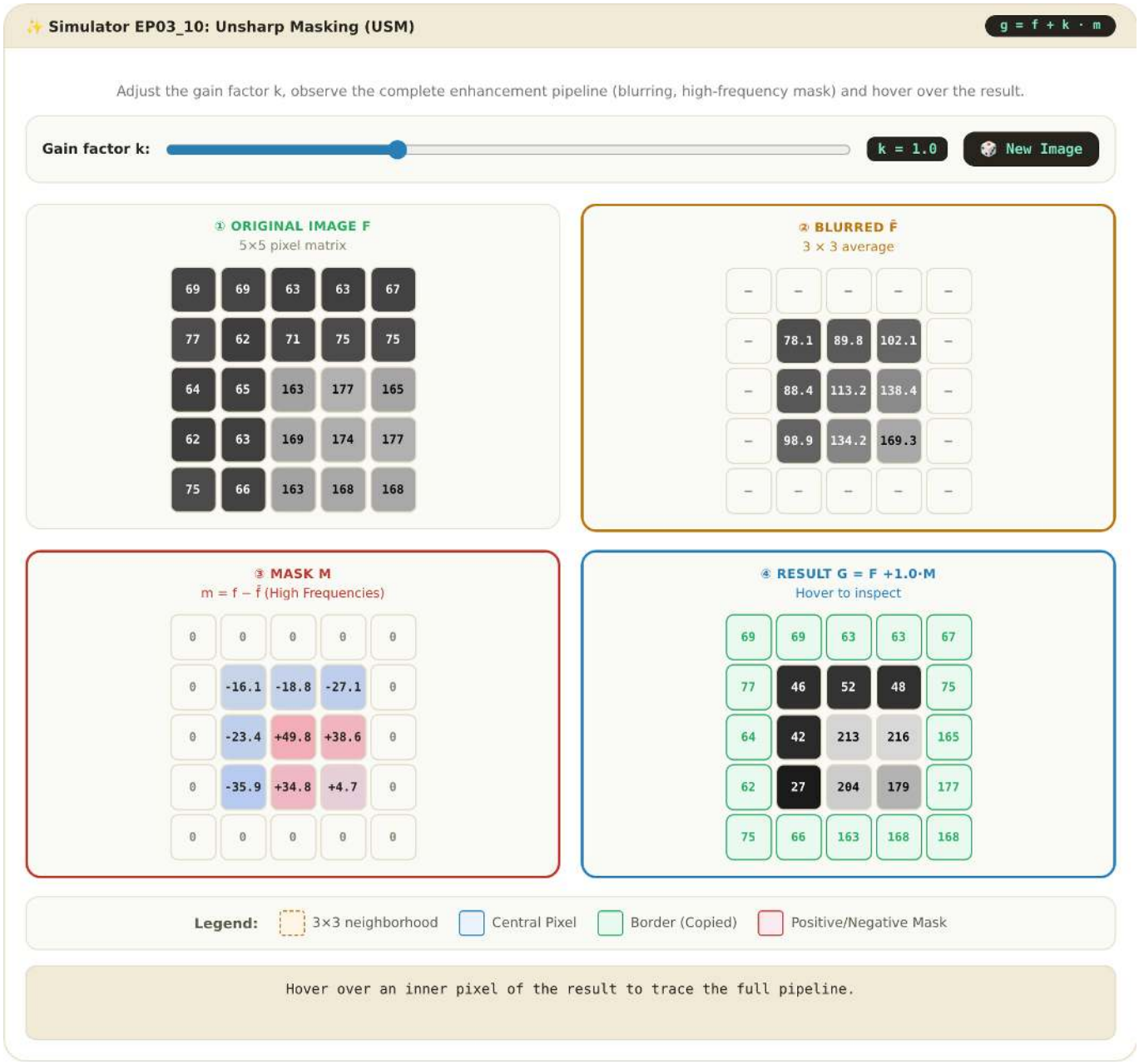
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Real k.
- Subsequent lines: Elements of the original matrix.

Output:

- Enhanced matrix $L \times C$.

3.12.10.5 📌 Examples

Input	Output	Observation
3 3 0.0 100 100 100 100 100 100 100 100 100	100 100 100 100 100 100 100 100 100	k=0: no enhancement
3 3 1.0 50 50 50 50 200 50 50 50 50	50 50 50 50 255 50 50 50 50	k=1: center pixel enhanced and saturated



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap03/sim-ep0310-unsharp.html

Figure 3.35: EP03_10 Simulator: *Unsharp Masking* (USM)

```
%%writefile EP03_10.cpp
// your solution
```

```
Overwriting EP03_10.cpp
```

```
TestSuite("EP03_10.cpp").run()
```

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

Chapter 4

Mathematical Morphology and Image Segmentation

This chapter presents two fundamental topics in Digital Image Processing (DIP): **mathematical morphology** and **image segmentation**. Mathematical morphology provides a theoretical framework based on set theory for analyzing, refining, and quantifying the shape of objects in binary and grayscale images, through fundamental operators such as **erosion** and **dilation**. Segmentation, in turn, aims to partition the image into regions of interest, separating objects from the background and producing suitable representations for analysis and interpretation.

The chapter begins with **thresholding**, one of the most important segmentation techniques, introducing the automatic **Otsu** method and revisiting histogram analysis through interclass variance, as presented in Chapter 1. Next, the main mathematical morphology operators are studied, including erosion, dilation, opening, closing, and morphological reconstruction, which allow refining binary masks and preserving relevant object structures. Finally, region-based segmentation techniques are presented, such as **connected component labeling**, the **distance transform**, and the marker-controlled ***watershed*** algorithm, culminating in the extraction of geometric descriptors and the generation of *bounding boxes* compatible with modern object detection systems.

4.1 Objectives

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

- **Apply thresholding:** Understand Otsu's automatic criterion by maximizing inter-class variance (σ_B^2) and select appropriate preprocessing strategies to facilitate segmentation;
- **Master binary morphology:** Understand and apply erosion ($A \ominus B$) and dilation ($A \oplus B$) as fundamental operators, deriving opening ($A \circ B$), closing ($A \bullet B$), and operations based on morphological reconstruction, such as `mm::clohole` and `mm::edgeoff`;
- **Apply grayscale morphology:** Use morphological gradient and *top-hat* filters for enhancement and analysis of local structures;
- **Label connected components:** Identify and separate connected regions in binary images through labeling algorithms;
- **Apply distance transform:** Interpret and compute distances to the background using morphological approaches and geometric metrics;
- **Segment by regions:** Build marker-based segmentation *pipelines* using Distance Transform and the ***watershed*** algorithm;
- **Extract geometric descriptors:** Calculate properties such as area, perimeter, centroid, circularity, and *bounding boxes* through `mm::label0` and contour extraction;
- **Relate DIP and computer vision:** Understand how descriptors extracted by segmentation can be converted to formats used by modern detectors, such as YOLO.

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # C++ track build artifacts (.cpp, binary, PNGs)

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")

# The kernel is Python even on the C++ track: `mm` (morph.py) is used by the
# simulators, the display of the figures the C++ binary generates, and the
# mm::Image state between cells. cpp=True also downloads the compiled track
# (morph.hpp + stb_image*.h), used in the #include of the %%writefile *.cpp cells.
# The C++ cells in this chapter compile WITH OpenCV (-DMM_USE_OPENCV +
# pkg-config opencv4): mm::dil/ero → cv::dilate/erode, so open/close/asf/
# gradm/tophat/blackhat run full-res and match bit-for-bit with the py track.
import config
config.setup(cpp=True)
from morph import mm
import numpy as np
```

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

4.2 Thresholding

Thresholding is one of the simplest and most efficient forms of image segmentation. Its goal is to classify each pixel into two intensity classes, typically associated with *object* and *background*:

$$g(x,y) = \begin{cases} 255, & \text{if } f(x,y) > T \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

where $f(x,y)$ represents the pixel intensity in the original image and $g(x,y)$ is the resulting binary image.

The choice of the threshold T is important for the quality of the segmentation. The **Otsu** method automatically determines the optimal threshold by maximizing the **between-class variance** σ_B^2, defined by:

$$\sigma_B^2(T) = w_0(T)\, w_1(T)\, \left[\mu_0(T) - \mu_1(T)\right]^2 \tag{4.2}$$

where:

- $w_0(T)$ and $w_1(T)$ are the cumulative probabilities of the background and object classes;
- $\mu_0(T)$ and $\mu_1(T)$ are the mean intensities of these classes;
- $\sigma_B^2(T)$ represents the between-class variance for a given threshold T.

The method works best when the histogram presents two relatively separated groups of intensities. To that end, the algorithm evaluates all possible thresholds of the image — typically in the interval $[0, 255]$ for 8-bit images — and selects the value that maximizes the between-class variance, denoted by σ_B^2:

$$T^* = \arg\max_{T \in [0,255]} \sigma_B^2(T)$$

i Otsu assumes bimodal histograms

The Otsu method yields better results when the histogram presents two well-defined peaks (*bimodality*), corresponding to the background and the object. The greater the separation between these peaks and the more pronounced the maximum of σ_B^2, the more reliable the obtained threshold tends to be.

In images with non-uniform illumination or multiple intensity regions, **adaptive thresholding** techniques — in which the threshold is computed locally — often produce more robust segmentations. The subscript B in σ_B^2 stands for ***between classes***. Thus, σ_B^2 represents the **between-class variance**.

4.2.1 Image of Coins

The image used to practice segmentation is a photograph of a collection of coins from different countries and eras (Figure 4.1). Credit: GAZI.MD.AHAD (CC BY-SA 4.0). It features circular objects with well-defined edges, making it ideal for demonstrating thresholding, morphological operators, distance transform, *watershed*, and shape descriptors.

```
%%writefile tmp/fig_04_coins.cpp
#define MM_OUT "tmp/fig_04_coins.png"
#include "morph.hpp"
#include <iostream>
#include <filesystem>

//| label: fig-04-coins
//| fig-cap: "Image with coins of various types. Credit: GAZI.MD.AHAD (CC BY-SA 4.0)."
//| echo: true

// The image is already in the chapter directory (imagens/coins.png); the Python
// track handles download/cache when the notebook runs standalone.
int main() {
    mm::Image img_coins_color = mm::read("imagens/coins.png");
    mm::Image img_coins_gray  = mm::gray(img_coins_color);

    std::cout << "Dimensions [y,x,c]: " << img_coins_color.h << ", "
              << img_coins_color.w << ", " << img_coins_color.channels << "\n";
    mm::show(img_coins_color, MM_OUT);

// [pdi:state-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp/state");
mm::write(img_coins_gray, "tmp/state/img_coins_gray_8.png");
// [pdi:state-io:end]
return 0;
}
```

```
Overwriting tmp/fig_04_coins.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_coins.cpp -o
tmp/fig_04_coins -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_coins \
  && test -f "tmp/fig_04_coins.png" \
  || echo "  mm::show não gravou tmp/fig_04_coins.png"
```

```
Dimensions [y,x,c]: 2560, 1920, 3
```

```
try:
    mm.show(mm.read("tmp/fig_04_coins.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_04_coins.png (ver
    a versao Python)")
```

Figure 4.1: Imagem com moedas de vários tipos. Crédito: GAZI.MD.AHAD (CC BY-SA 4.0).

4.2.2 Preprocessing for Otsu

Otsu's method relies on a well **bimodal** histogram. The coin image has non-uniform illumination and dark coins close to the background, which hinders this. In the C++ track, **global histogram equalization** (`mm::equalize`) is applied before thresholding — the CLAHE × Gaussian comparison and the $\sigma_B^2(T)$ curves are left for the Python track (Figure 4.2).

```
%%writefile tmp/fig_04_otsu_comparacao_histogramas.cpp
#define MM_OUT "tmp/fig_04_otsu_comparacao_histogramas.png"
//| label: fig-04-otsu-comparacao-histogramas
//| fig-cap: "CLAHE (adaptive contrast-limited enhancement) before Otsu thresholding:
original, enhanced, histogram and binarization. (The Python track also plots the ²_B(T)
curves.)"
//| echo: true
//| output: true
```

```
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_coins_gray = mm::_read_state("tmp/state/img_coins_gray_8.png");
// [pdi:state-io:end]

    // img_coins_gray is provided already initialized

    mm::Image img_clahe = mm::clahe(img_coins_gray, 2.0, 8);   // adaptive contrast-limited
enhancement
    mm::Image img_gauss = mm::gaussian(img_clahe, 5, 0);

    mm::show(
        std::vector<mm::Image>{img_coins_gray, img_clahe, mm::histImg(img_clahe),
mm::threshold(img_clahe)},
        MM_OUT,
        std::vector<std::string>{"Original", "CLAHE", "Histograma (CLAHE)", "Otsu"},
        4
    );


// [pdi:state-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp/state");
mm::write(img_clahe, "tmp/state/img_clahe_12.png");
// [pdi:state-io:end]
return 0;
}
```

```
Overwriting tmp/fig_04_otsu_comparacao_histogramas.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4
tmp/fig_04_otsu_comparacao_histogramas.cpp -o tmp/fig_04_otsu_comparacao_histogramas
-lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_otsu_comparacao_histogramas \
  && test -f "tmp/fig_04_otsu_comparacao_histogramas.png" \
  || echo "  mm::show não gravou tmp/fig_04_otsu_comparacao_histogramas.png"
```

```
[1] Original
[2] CLAHE
[3] Histograma (CLAHE)
[4] Otsu
```

```
try:
    mm.show(mm.read("tmp/fig_04_otsu_comparacao_histogramas.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_otsu_comparacao_histogramas.png (ver a versao Python)")
```

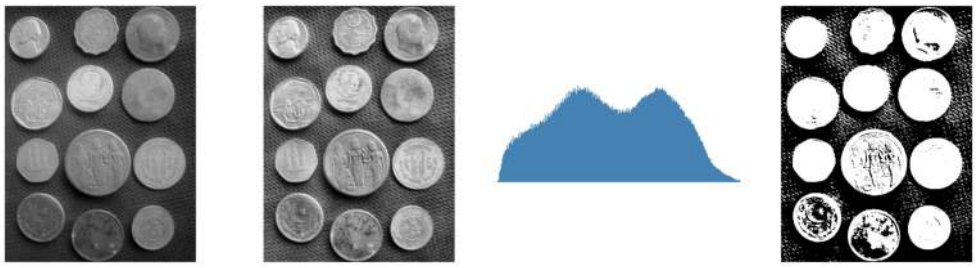


Figure 4.2: CLAHE (realce adaptativo com limite de contraste) antes da limiarização de Otsu: original, realçado, histograma e binarização. (A trilha Python também traça as curvas de $\sigma_B^2(T)$.)

4.2.3 Result: CLAHE as the Best Pre-processing

The analysis of Figure 4.2 indicates that **CLAHE** achieved the highest inter-class variance ($\sigma_B^2 \approx 2{,}47 \times 10^3$), with an optimal threshold $T^* = 122$. Although the **CLAHE+Gaussian** combination produced a very similar result ($\sigma_B^2 \approx 2{,}43 \times 10^3$, $T^* = 123$), the quantitative criterion of Otsu's method slightly favors the use of CLAHE alone.

In visual terms, the binarized images obtained with CLAHE and CLAHE+Gaussian are practically equivalent. The difference between the two approaches becomes more evident in the analysis of the histograms and the values of $\sigma_B^2(T)$ than in the direct inspection of the resulting segmentations. Thus, the choice of CLAHE is mainly based on maximizing the statistical separation between the background and object classes.

Interpretation of the results

Note that the CLAHE and CLAHE+Gaussian pre-processings produce very similar histograms and optimal thresholds ($T^* = 122$ and $T^* = 123$). Consequently, the resulting binarized images are also quite similar. In this case, the decision is not based on striking visual differences but on the objective criterion of Otsu's method: the highest value of σ_B^2 indicates the best separation between the classes.

4.3 Mathematical Morphology

Mathematical morphology is a set-based theory used to analyze the shape and structure of objects in images. Unlike the linear filters presented in Chapter 3, morphological operators are **nonlinear**, as they are based on minimum, maximum, and spatial inclusion operations, rather than linear combinations of intensities. These operators act on the neighborhood of each pixel through a **structuring element** $\mathbb{B}$, which defines the shape and size of the analyzed region.

In binary and grayscale images with flat elements, the structuring element translated to position x is defined spatially as:

$$\mathbb{B}_x = \{x + b \mid b \in \mathbb{B}\}$$

At image border regions, part of the set $\mathbb{B}_x$ may extend beyond the physical domain of the scene ($\mathbb{E}$). To ensure the mathematical consistency of the primitive operators at these boundaries, it is theoretically assumed that the space exterior to the image domain is filled with the **neutral element** of the corresponding operation (positive infinity for erosion and negative infinity for dilation), preventing the external environment from corrupting the internal structures of the object.

When the structuring element associates weights to its elements — that is, $b : \mathbb{B} \to \mathbb{Z}$ — it is termed a **structuring function** or **non-flat structuring element**.

Developed by Matheron and Serra in the 1960s for binary images and subsequently extended to grayscale, mathematical morphology underpins operators such as morphological gradient, *top-hat*, *watershed*, and distance transform, all derived from two primitives: **erosion** and **dilation** [Matheron (1975); Serra (1982)].

4.3.1 Erosion and Dilation

The two primitive operators are defined in a unified manner for grayscale images ($f : \mathbb{E} \to \mathbb{Z}$) and, by restriction to the domain $\{0, 1\}$, also for binary images.

4.3.1.1 Erosion

The **erosion** of an image f by a structuring function $b : \mathbb{B} \to \mathbb{Z}$ is formally defined by:

$$\varepsilon_b(f)(x) = (f \ominus b)(x) = \min_{z \in \mathbb{B}} \{ f(x + z) - b(z) \}, \quad \forall\, x \in \mathbb{E} \tag{4.3}$$

In practice, erosion replaces the intensity of pixel x with the minimum value resulting from the difference between the image and the structuring element in the neighborhood defined by the domain $\mathbb{B}$. Positive values in the weights of $b(z)$ force the local result downward, "digging" the image relief more deeply and intensifying the erosion.

In the **flat** case (where the weights are null within the domain, i.e., $b \equiv 0$), the expression simplifies to the pure local minimum:

$$\varepsilon_B(f)(x) = \min\{ f(y) : y \in \mathbb{B}_x \}$$

In binary images, this operation is equivalent to requiring that the set $\mathbb{B}$, translated to coordinate x, be **completely contained** in the object A:

$$A \ominus \mathbb{B} = \{ z \in \mathbb{E} \mid \mathbb{B}_z \subseteq A \}$$

Visual Effect: *Shrinks* bright objects and structures, eliminating protrusions, bright peaks, or noise that are geometrically smaller than the domain $\mathbb{B}$.

4.3.1.2 Implementation of erosion

The didactic version `mm::ero0` implements the particular case of erosion with a **flat structuring element**. For each pixel (y, x), the function traverses the spatial neighbors allowed by B and stores the smallest value found in the input image f, directly reproducing the local minimum operation described in Equation 4.3 for $b \equiv 0$.

Note that the neighbor values are always read statically from the original image f; the output matrix g is used exclusively to record the accumulated minimum of the current neighborhood. Thus, the final result is invariant with respect to the pixel scanning order (whether by rows or columns).

The auxiliary function `_viz` computes the coordinates of valid neighbors within the physical boundaries of the image. At the borders, the initialization of the accumulator to `255` exactly emulates the padding with the neutral element required by the theory. The interface function `mm::ero`, in turn, resorts to the native and optimized OpenCV implementation (`mm::ero`) when the structuring element is flat, switching to the general routine `mm::ero1` if the element has topographic weights.

The following computational example illustrates the application of a cross-shaped structuring element (`mm::secross()`) highlighted in Figure 4.3, comparing the execution of the didactic loop-based variant (`mm::ero0`) with the computational engine of OpenCV (`mm::ero`).

```
%%writefile tmp/fig_04_elemento_cruz.cpp
#define MM_OUT "tmp/fig_04_elemento_cruz.png"
//| label: fig-04-elemento-cruz
//| fig-cap: "Elemento estruturante em formato de cruz ($B_{\\text{cruz}}$) utilizado para
conectividade-4."
//| echo: true
//| output: true

#include "morph.hpp"

int main() {
    mm::Image B_cruz = mm::secross();
    mm::drawImgPlt(B_cruz, MM_OUT, 40);
    return 0;
}
```

```
Overwriting tmp/fig_04_elemento_cruz.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_elemento_cruz.cpp -o
tmp/fig_04_elemento_cruz -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_elemento_cruz \
  && test -f "tmp/fig_04_elemento_cruz.png" \
  || echo "  mm::show não gravou tmp/fig_04_elemento_cruz.png"
```

```
0 1 0
1 1 1
0 1 0
```

```
try:
    mm.show(mm.read("tmp/fig_04_elemento_cruz.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_elemento_cruz.png (ver a versao Python)")
```

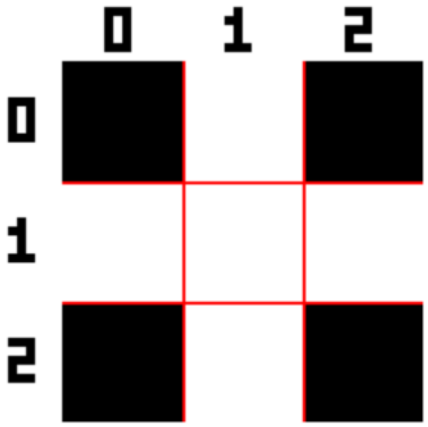


Figure 4.3: Elemento estruturante em formato de cruz (B_{cruz}) utilizado para conectividade-4.

```
# The morph.hpp is header-only; below, the _viz helper and the body of mm::ero0().
import re, pathlib
hpp = pathlib.Path("morph.hpp").read_text()
for nome, pat in [("_viz", r"template <class F>\ninline void _viz\(.*?\n\}"),
                  ("ero0", r"inline Image ero0\(.*?\n\}")]:
    m = re.search(pat, hpp, re.S)
```

```
    print(f"// ---- mm::{nome} ----")
    print(m.group(0) if m else f"({nome} não encontrada)")
    print()
```

```
// ---- mm::_viz ----
template <class F>
inline void _viz(const Image& f, const SE& B, int y, int x, F&& cb) {
    double oh = -B.h / 2.0 + 0.5;
    double ow = -B.w / 2.0 + 0.5;
    for (int by = 0; by < B.h; ++by)
        for (int bx = 0; bx < B.w; ++bx) {
            int vy = (int)(y + by + oh);
            int vx = (int)(x + bx + ow);
            if (vy >= 0 && vy < f.h && vx >= 0 && vx < f.w)
                cb(vy, vx, B.at(by, bx));
        }
}

// ---- mm::ero0 ----
inline Image ero0(const Image& f, SE Bc = SE::box(3)) {
    _require_gray(f, "ero0");
    Image g(f.h, f.w, 1);
    for (int y = 0; y < f.h; ++y)
        for (int x = 0; x < f.w; ++x) {
            int mn = 255;
            _viz(f, Bc, y, x, [&](int vy, int vx, int bv) {
                if (bv != 0 && (int)f.at(vy, vx) < mn) mn = f.at(vy, vx);
            });
            g.at(y, x) = (unsigned char)mn;
        }
    return g;
}
```

4.3.1.3 Dilation

Dilation of an image f by a structuring function $b : \mathbb{B} \to \mathbb{Z}$ is formally defined by:

$$\delta_b(f)(x) = (f \oplus b)(x) = \max_{z \in \mathbb{B}} \{ f(x - z) + b(z) \}, \quad \forall\, x \in \mathbb{E} \tag{4.4}$$

In practice, dilation replaces the intensity of pixel x with the largest value resulting from the sum between the image and the structuring element within the defined neighborhood. The spatial inversion argument $(x - z)$ indicates that dilation implicitly evaluates the transposed (reflected) element $\hat{b}$, a fundamental property for ensuring mathematical duality with respect to erosion.

In the **flat** case (where the weights are null within the domain, i.e., $b \equiv 0$), the expression reduces to pure local maximum:

$$\delta_B(f)(x) = \max\{ f(y) : y \in \mathbb{B}_x \}$$

For binary images, this operation is equivalent to requiring that the reflected set $\hat{\mathbb{B}}$, translated to coordinate x, has a **non-empty intersection** with object A:

$$A \oplus \mathbb{B} = \{ z \in \mathbb{E} \mid \hat{\mathbb{B}}_z \cap A \neq \emptyset \}$$

Visual Effect: *Expands* the bright structures of the image, increasing object filling, connecting nearby components, and eliminating channels, dark pits, or valleys that are geometrically smaller than the domain $\mathbb{B}$.

4.3.1.4 Implementation of dilation

The didactic version `mm::dil0` implements the particular case of dilation with a **flat structuring element**. For each pixel (y, x), the function traverses the spatial neighbors allowed by B and stores the largest value found in the input image f, directly reproducing the local maximum operation for $b \equiv 0$.

Before starting the spatial scan, the structuring element undergoes a geometric reflection through the `np.flip(Bc)` instruction to explicitly construct the transposed matrix $\hat{B}$ required by the theory. In perfectly symmetric masks (such as crosses, squares, and disks centered at the origin), this reflection does not alter the pixel arrangement; however, for asymmetric elements, this step is strictly necessary to ensure equivalence with the formal definitions and to safeguard the duality laws.

As verified in the erosion operator, neighbor values are always read statically from the original matrix f, while the output matrix g acts purely as the register of the accumulated neighborhood maximum. At the image boundaries, initializing the accumulator to `0` exactly emulates the external padding with the neutral element ($-\infty$, or zero in 8-bit representations), ensuring that the physical edges of the scene are dilated in perfect conformity with the standard adopted by OpenCV.

The computational example below illustrates the practical application of a cross-shaped element (`mm::secross()`), validating the consistency between the loop-based logic (`mm::dil0`) and the native industrial method (`mm::dil`).

```
# Body of mm::dil0() in morph.hpp (reflects the SE before scanning, like np.flip).
import re, pathlib
hpp = pathlib.Path("morph.hpp").read_text()
m = re.search(r"inline Image dil0\(.*?\n\}", hpp, re.S)
print(m.group(0) if m else "(dil0 não encontrada)")
```

```
inline Image dil0(const Image& f, SE Bc = SE::box(3)) {
    _require_gray(f, "dil0");
    SE B = Bc.reflected();
    Image g(f.h, f.w, 1);
    for (int y = 0; y < f.h; ++y)
        for (int x = 0; x < f.w; ++x) {
            int mx = 0;
            _viz(f, B, y, x, [&](int vy, int vx, int bv) {
                if (bv != 0 && (int)f.at(vy, vx) > mx) mx = f.at(vy, vx);
            });
            g.at(y, x) = (unsigned char)mx;
        }
    return g;
}
```

i Note: The Sign Confrontation ($f(x+z)$ vs $f(x-z)$)

Compare the formal definitions of erosion (Equation 4.3) and dilation (Equation 4.4). Consider an asymmetric structuring element to the right $\mathbb{B} = \{0, 1\}$ (origin and one pixel to the right) applied at position $x = 10$.

1. **In Erosion** (Equation 4.3):

$$\min\{f(x+z) - b(z)\}$$

   - $z = 0 \Rightarrow f(10+0) = \mathbf{f(10)}$
   - $z = 1 \Rightarrow f(10+1) = \mathbf{f(11)}$

   The operator queries the current pixel (10) and the pixel to the right (11), preserving the original orientation of $\mathbb{B}$.
2. **In Dilation** (Equation 4.4):

$$\max\{f(x-z)+b(z)\}$$

- $z=0 \Rightarrow f(10-0)=\mathbf{f(10)}$
- $z=1 \Rightarrow f(10-1)=\mathbf{f(9)}$

Due to the negative sign $(-z)$, advancing in the structuring element corresponds to moving backward in the image, causing dilation to query the pixel to the left (9).

The `_viz` function, used in `morph.py`, generates neighbors by additive shifts of the form $x+z$. For this reason, the implementation of `mm::dil0` previously reflects the structuring element using `np.flip(B)`. After reflection, the scan based on $x+z$ accesses exactly the same points defined by the theoretical expression $f(x-z)$ of dilation in Equation 4.4.

For symmetric structuring elements (such as discs, squares, and centered crosses), reflection does not alter the mask. For asymmetric elements, however, this step is essential for the implementation to correctly reproduce the mathematical definition of dilation and preserve the erosion–dilation duality.

i Erosion–dilation duality

Erosion and dilation are **dual by complement**. This means that one operator can be fully obtained from the other, provided that one operates on the complement of the image using the reflected structuring element $\hat{B}$:

$$(A \ominus B)^c = A^c \oplus \hat{B} \quad \Longleftrightarrow \quad A \ominus B = (A^c \oplus \hat{B})^c$$

Similarly, **dilation can also be obtained from erosion**:

$$(A \oplus B)^c = A^c \ominus \hat{B} \quad \Longleftrightarrow \quad A \oplus B = (A^c \ominus \hat{B})^c$$

In practical terms, the erosion of an object can be obtained by dilating its complement, followed by complementing the result (and vice versa). In the implementation of the `morph.py` package, the didactic versions `mm::ero0` and `mm::dil0` make this structure explicit through loops, while `mm::ero` and `mm::dil` delegate the operations to OpenCV for greater computational efficiency.

i Boundary conditions and finite images

In classical mathematical morphology, defined over an infinite domain (typically $\mathbb{Z}^2$), this duality is exact. In digital images, however, one works with finite matrices, and the result depends on how pixels located outside the image are treated.

For the duality identities to remain valid, the complement must be defined relative to the same universe, and the boundary conditions adopted for erosion and dilation must be complementary to each other. For example, if erosion assumes that external pixels belong to the object (255), then dilation applied to the complement must assume that these same external pixels belong to the background (0).

When different filling strategies are used (replication, reflection, constant value, etc.), the theoretical duality may no longer be satisfied exactly in regions near the image borders.

To numerically illustrate the morphological operators and the erosion–dilation duality, Figure 4.4 presents a 10×10 binary image processed with an "L"-shaped structuring element. In the implementation of `morph.py`, the origin of B is set at the geometric center of the mask — position $(1,1)$ for a 3×3 *kernel* — and must correspond to an active element for the erosion to behave correctly (as discussed earlier). The structuring element B_L defined below satisfies this condition. Figure 4.5 complements the analysis with an interactive erosion simulator, allowing visualization of the displacement of the structuring element over the image and identification of the positions where it remains completely contained within the object.

```
%%writefile tmp/fig_04_ero_dil_didatico.cpp
#define MM_OUT "tmp/fig_04_ero_dil_didatico.png"
```

```cpp
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <filesystem>

int main() {
    //| label: fig-04-ero-dil-didatico
    //| fig-cap: "Erosão e dilatação em imagem binária 10×10 com elemento estruturante 'L'
    assimétrico 3×3. Validação da dualidade erosão-dilatação."
    //| echo: true
    //| output: true

    mm::Image A(10, 10);
    // Fill the 10x10 image with the pattern (values are 0 or 255)
    unsigned char pattern[10][10] = {
        {0,0,0,0,0,0,0,0,0,0},
        {0,0,0,255,255,255,0,0,0,0},
        {0,0,255,255,255,255,255,0,0,0},
        {0,255,255,255,255,255,255,255,0,0},
        {0,255,255,255,255,255,255,255,0,0},
        {0,255,255,255,255,255,255,0,0,0},
        {0,0,255,255,255,255,255,255,0,0},
        {0,0,0,255,255,255,255,0,0,0},
        {0,0,0,0,255,0,0,0,0,0},
        {0,0,0,0,0,0,0,0,0,0}
    };
    for (int y = 0; y < 10; y++)
        for (int x = 0; x < 10; x++)
            A.at(y, x) = pattern[y][x];

    mm::SE B_L{{1,0,0},{1,1,0},{1,1,0}};   // 'L', origem no centro
    mm::SE B_hat{{0,1,1},{0,1,1},{0,0,1}};   // B_L girado 180° (B)

    std::cout << "Elemento estruturante B_L:" << std::endl;
    mm::Image B_L_img = mm::sebox(0);
    // Create the B_L binary image for display
    B_L_img = mm::Image(3, 3);
    B_L_img.at(0,0) = 0; B_L_img.at(0,1) = 255; B_L_img.at(0,2) = 255;
    B_L_img.at(1,0) = 0; B_L_img.at(1,1) = 255; B_L_img.at(1,2) = 255;
    B_L_img.at(2,0) = 0; B_L_img.at(2,1) = 0; B_L_img.at(2,2) = 255;
    std::cout << mm::drawImg(B_L_img) << std::endl;

    std::cout << "Elemento estruturante refletido B:" << std::endl;
    mm::Image B_hat_img(3, 3);
    B_hat_img.at(0,0) = 255; B_hat_img.at(0,1) = 0; B_hat_img.at(0,2) = 0;
    B_hat_img.at(1,0) = 255; B_hat_img.at(1,1) = 255; B_hat_img.at(1,2) = 0;
    B_hat_img.at(2,0) = 255; B_hat_img.at(2,1) = 255; B_hat_img.at(2,2) = 0;
    std::cout << mm::drawImg(B_hat_img) << std::endl;

    mm::Image img_ero0 = mm::ero0(A, B_L);

    // Dualidade: (A   B)  == A    B
    mm::Image A_c    = mm::neg(A);
    mm::Image ero_c  = mm::neg(img_ero0);
    mm::Image dil_Ac = mm::dil0(A_c, B_hat);

    bool equal = true;
    for (int i = 0; i < ero_c.h * ero_c.w; i++) {
        if (ero_c.data[i] != dil_Ac.data[i]) {
            equal = false;
            break;
```

```
        }
    }
    std::cout << "Dualidade (A  B) == A  B : " << (equal ? "True" : "False") << std::endl;

    mm::show(
        std::vector<mm::Image>{A, A_c, img_ero0, ero_c, dil_Ac},
        MM_OUT,
        std::vector<std::string>{"A", "A ", "A  B", "(A  B) ", "A  B"},
        5
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(A, "tmp/fig_04_ero_dil_didatico_0.png");
mm::write(A_c, "tmp/fig_04_ero_dil_didatico_1.png");
mm::write(img_ero0, "tmp/fig_04_ero_dil_didatico_2.png");
mm::write(ero_c, "tmp/fig_04_ero_dil_didatico_3.png");
mm::write(dil_Ac, "tmp/fig_04_ero_dil_didatico_4.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_04_ero_dil_didatico.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_ero_dil_didatico.cpp -o
tmp/fig_04_ero_dil_didatico -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_ero_dil_didatico \
  && test -f "tmp/fig_04_ero_dil_didatico.png" \
  || echo "  mm::show não gravou tmp/fig_04_ero_dil_didatico.png"
```

```
Elemento estruturante B_L:
  0 255 255
  0 255 255
  0   0 255

Elemento estruturante refletido B:
255   0   0
255 255   0
255 255   0

Dualidade (A  B) == A  B : True
[1] A
[2] A
[3] A  B
[4] (A  B)
[5] A  B
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_04_ero_dil_didatico_0.png"),
            mm.read("tmp/fig_04_ero_dil_didatico_1.png"),
            mm.read("tmp/fig_04_ero_dil_didatico_2.png"),
            mm.read("tmp/fig_04_ero_dil_didatico_3.png"),
            mm.read("tmp/fig_04_ero_dil_didatico_4.png"),
        ],
        titles=[
            'A',
            'A ',
```

```
            'A   B',
            '(A   B) ',
            'A   B',
        ],
        cols=5,
        figsize=(15, 3),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_ero_dil_didatico_0.png (ver a versao Python)")
```

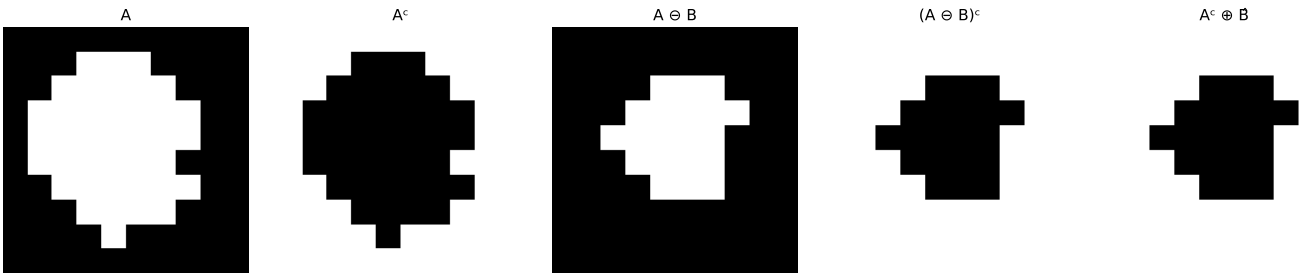
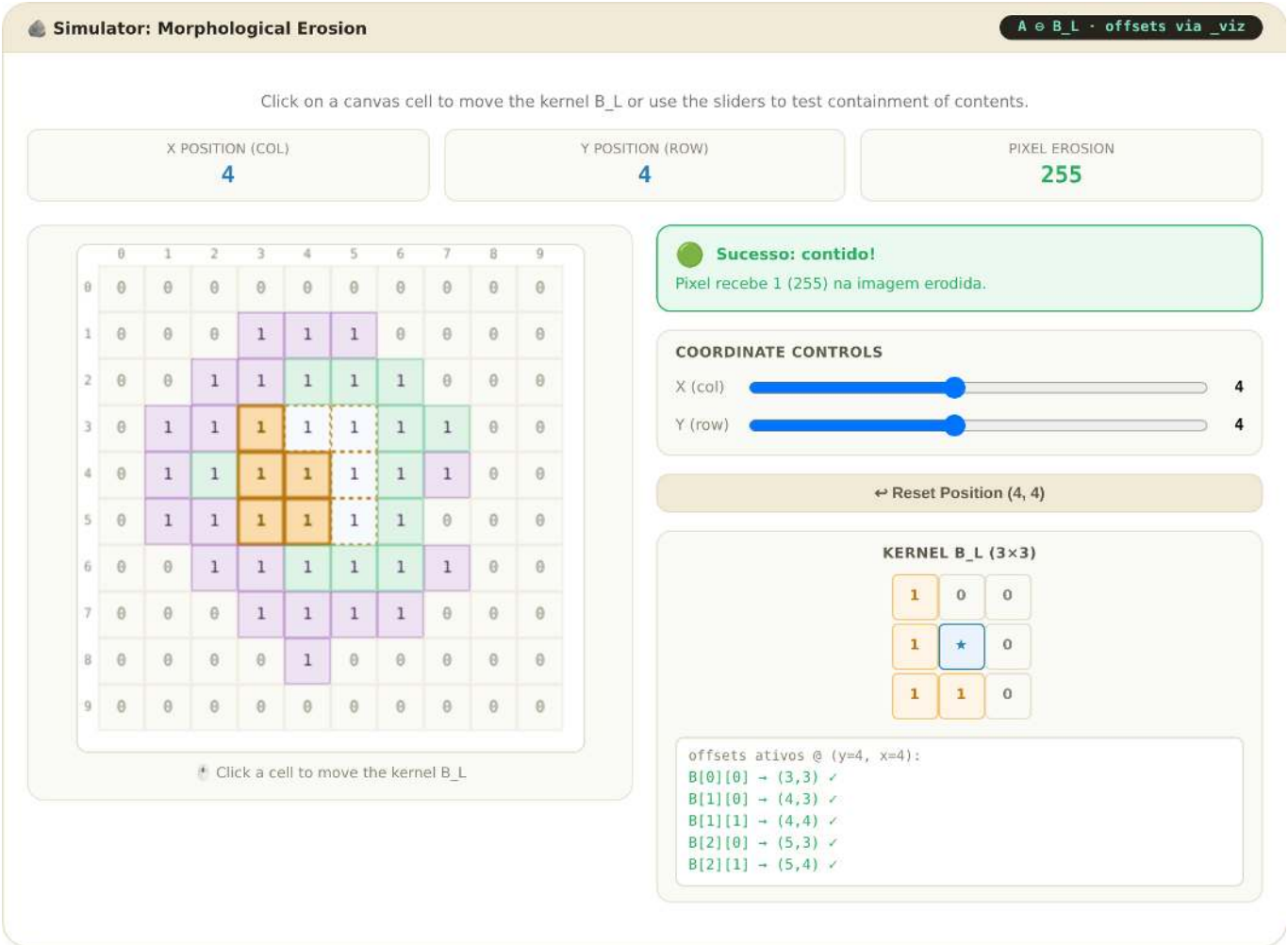


Figure 4.4: Erosão e dilatação em imagem binária 10×10 com elemento estruturante ‘L’ assimétrico 3×3. Validação da dualidade erosão–dilatação.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-04-erosao.html

Figure 4.5: Simulator: Morphological Erosion (A B_L)

4.3.2 Opening and Closing

Combining erosion and dilation yields two operators of great practical utility: **opening** and **closing**, defined by Equations Equation 4.5 and Equation 4.6. Their main effects are summarized in Table 4.1.

Opening (*opening*) — erosion followed by dilation using the same B:

$$A \circ B = (A \ominus B) \oplus B \tag{4.5}$$

Closing (*closing*) — dilation followed by erosion using the same B:

$$A \bullet B = (A \oplus B) \ominus B \tag{4.6}$$

In practice, `mm::open` and `mm::close` apply the same structuring element in both stages. For symmetric structuring elements (the most common ones), this implementation coincides with the mathematical definition presented above.

Table 4.1: Properties of opening and closing.

Operator	Sequence	Main effect
Opening $A \circ B$	erosion → dilation	Removes structures unable to contain the structuring element; smooths external contours
Closing $A \bullet B$	dilation → erosion	Fills holes smaller than B; smooths internal contours

Important property: both are **idempotent**. For example,

$$(A \circ B) \circ B = A \circ B,$$

that is, after the first application, further applications of the same operator no longer alter the result.

4.3.2.1 Implementation of opening and closing

Unlike erosion and dilation, opening and closing do not introduce new computational mechanisms. Both are obtained by the sequential composition of the primitive operators already presented:

```
def open0(f, B):
    return mm.dil0(mm.ero0(f, B), B)

def close0(f, B):
    return mm.ero0(mm.dil0(f, B), B)
```

The `mm::open` function delegates the operation to `mm::open(f, B)`, while `mm::close` uses `mm::close(f, B)`, producing the same result more efficiently.

Opening inherits from erosion the ability to remove structures smaller than the structuring element and from dilation the partial restoration of preserved regions. Closing performs the reverse process: it first expands the objects and then restores their original dimensions, filling gaps and holes smaller than the structuring element.

4.3.2.2 Alternating Sequential Filter

In practice, opening and closing are often applied in sequence to simultaneously remove external noise and fill internal holes. The function `mm::asf` (*Alternating Sequential Filter*) generalizes this strategy by applying openings and closings alternately with progressively larger structuring elements. The available sequences are presented in Table 4.2.

Table 4.2: Sequences of the alternating sequential filter mm.asf.

Sequence	Order	Typical use
`'OC'`	opening → closing	removes external noise before filling small holes
`'CO'`	closing → opening	fills small holes before removing external noise
`'OCO'`	opening → closing → opening	emphasizes the removal of external noise
`'COC'`	closing → opening → closing	emphasizes the filling of holes and gaps

The parameter `n` controls the number of scales used by the filter. At each iteration i, the structuring element is enlarged by Minkowski addition (`mm::sesum(b, i)`), producing a sequence of increasingly comprehensive morphological filters. Unlike a single opening or closing with a large structuring element, the ASF performs progressive smoothing at multiple scales, better preserving the geometry of relevant objects while eliminating smaller structures. Figure 4.6 presents an example of applying opening, closing, and the ASF to the image of the coins.

```
%%writefile tmp/fig_04_open_close.cpp
#define MM_OUT "tmp/fig_04_open_close.png"
// Compile: g++ -std=c++17 -I. -o program program.cpp -lcurl -lpng -lz

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_clahe = mm::_read_state("tmp/state/img_clahe_12.png");
// [pdi:state-io:end]

    //| label: fig-04-open-close
    //| fig-cap: "Opening, closing, composition and alternating sequential filter applied to
    Otsu binarization of coins (pre-processed by contrast enhancement). Structuring element:
    13×13 disk."
    //| echo: true
    //| output: true

    mm::Image img_bin = mm::threshold(img_clahe);
    mm::Image B_disk = mm::sedisk(19);

    mm::Image img_open = mm::open(img_bin, B_disk);             // erosion → dilation
    mm::Image img_close = mm::close(img_bin, B_disk);            // dilation → erosion
    mm::Image img_oc = mm::close(img_open, B_disk);           // opening followed by closing
    mm::Image img_asf = mm::asf(img_bin, "OC", mm::sedisk(3), 7);
    // ASF: 3×3 base disk, grows each iteration

    mm::show(
        std::vector<mm::Image>{img_bin, img_open, img_close, img_oc, img_asf},
        MM_OUT,
        std::vector<std::string>{"Otsu Binarization", "Opening (A B)", "Closing (A B)",
                "Opening→Closing", "ASF-OC (n=7)"},
```

```
        5
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_bin, "tmp/fig_04_open_close_0.png");
mm::write(img_open, "tmp/fig_04_open_close_1.png");
mm::write(img_close, "tmp/fig_04_open_close_2.png");
mm::write(img_oc, "tmp/fig_04_open_close_3.png");
mm::write(img_asf, "tmp/fig_04_open_close_4.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_04_open_close.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_open_close.cpp -o
tmp/fig_04_open_close -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_open_close \
  && test -f "tmp/fig_04_open_close.png" \
  || echo "  mm::show não gravou tmp/fig_04_open_close.png"
```

```
[1] Otsu Binarization
[2] Opening (A B)
[3] Closing (A B)
[4] Opening→Closing
[5] ASF-OC (n=7)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_04_open_close_0.png"),
            mm.read("tmp/fig_04_open_close_1.png"),
            mm.read("tmp/fig_04_open_close_2.png"),
            mm.read("tmp/fig_04_open_close_3.png"),
            mm.read("tmp/fig_04_open_close_4.png"),
        ],
        titles=[
            'Binarização Otsu',
            'Abertura (A B)',
            'Fechamento (A B)',
            'Abertura→Fechamento',
            'ASF-OC (n=7)',
        ],
        cols=5,
        figsize=(15, 12),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_open_close_0.png (ver a versao Python)")
```

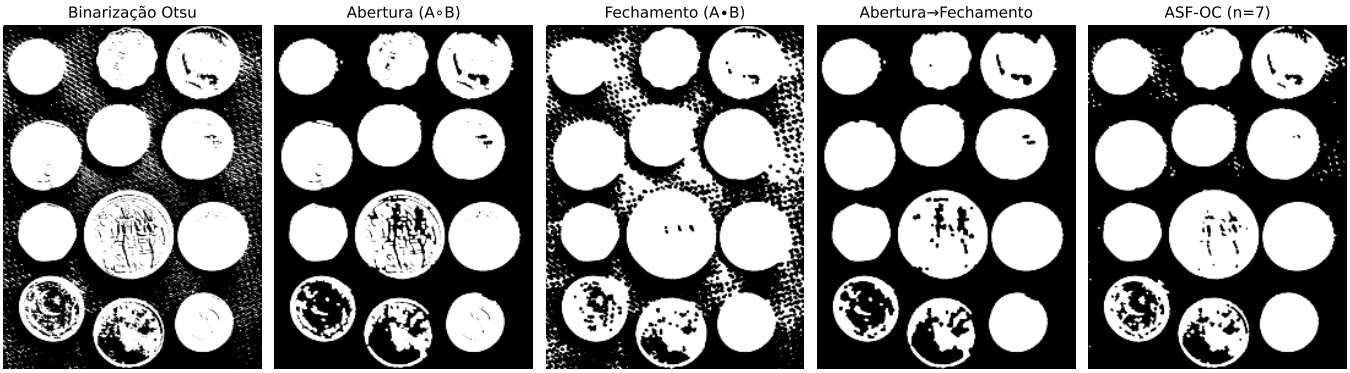


Figure 4.6: Abertura, fechamento, composição e filtro sequencial alternado aplicados à binarização Otsu das moedas (pré-processadas por realce de contraste). Elemento estruturante: disco 13×13.

4.3.3 Geodesic Operators

Geodesic operators introduce an additional constraint to classical morphological operators through a control image called the **mask** g. Instead of allowing erosion or dilation to propagate freely across the image, the result of each iteration is limited pointwise by the mask values, restricting the operation's evolution to permitted regions.

4.3.3.1 Geodesic Dilation

The **geodesic dilation** of a marker image f under a mask image g, using a flat structuring element b, is defined by:

$$f \oplus_g b = (f \oplus b) \wedge g, \tag{4.7}$$

where $\wedge$ represents the pointwise minimum.

In other words, a conventional dilation is initially performed on the marker, and then the result is constrained by the mask g. In this way, the propagation can never exceed the regions permitted by the mask.

The classical formulation of geodesic dilation assumes that the marker is contained within the mask, that is, $f \leq g$, ensuring that the evolution of the operation always remains bounded by the mask.

4.3.3.2 Implementation of geodesic dilation

The `mm::cdil` function directly implements this operator and allows executing multiple consecutive iterations:

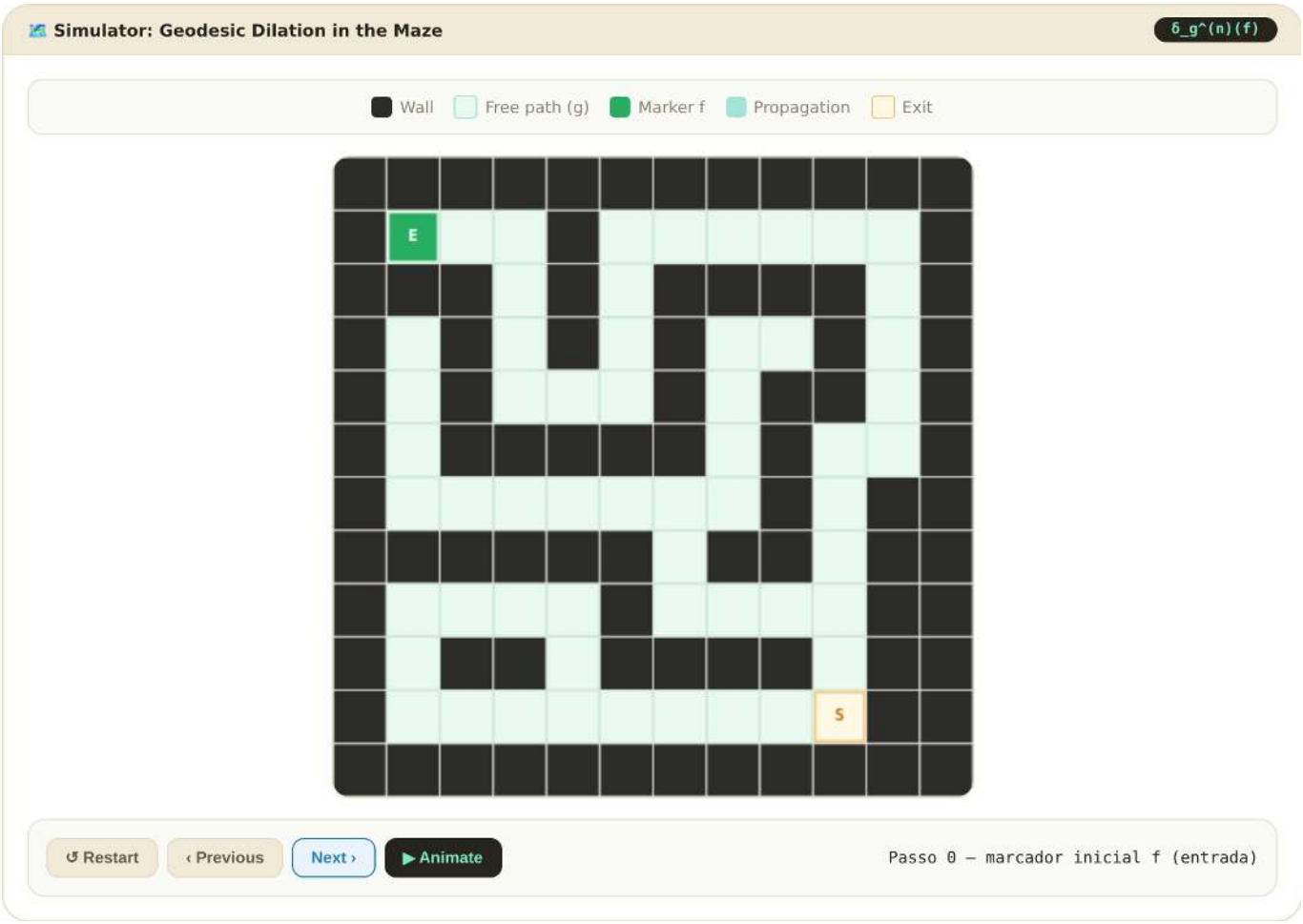
```
def cdil(f, g, b=np.zeros((3,3),dtype='uint8'), n=1):
    """Dilatação geodésica do marcador f sob a máscara g."""
    y = f.copy()
    for _ in range(n):
        y = np.minimum(mm.dil(y, b), g)
    return y
```

The `np.minimum(mm::dil(y, b), g)` instruction exactly implements the mathematical definition of geodesic dilation, that is, $(y \oplus b) \wedge g$.

When $n = 1$, the function performs a single geodesic dilation. For $n > 1$, the result of each step becomes the marker for the next step, producing a progressive propagation controlled by the mask.

The interactive simulator Figure 4.7 allows you to follow, step by step, the propagation of the marker f along the corridors of the maze. At each iteration of `mm::cdil`, the dilation front advances to the neighboring free cells — those where $g = 1$ — while the walls ($g = 0$) remain impassable. The number of steps required for

the marker to reach the exit corresponds exactly to the geodesic length of the shortest path within the mask, highlighting the direct connection between iterated geodesic dilation and the notion of distance in graphs.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-04-cdil.html

Figure 4.7: Interactive simulator of geodesic dilation: the marker f (green) propagates step by step through the free paths of mask g, without crossing walls.

4.3.3.3 Geodesic Erosion

Dually, the **geodesic erosion** of a marker image f under a mask image g is defined by:

$$f \ominus_g b = (f \ominus b) \vee g, \tag{4.8}$$

where $\vee$ represents the pointwise maximum operator.

In this case, the conventional erosion of the marker is followed by a lower constraint imposed by the mask. Thus, no pixel of the result can assume a value lower than the corresponding pixel of the mask.

The classical formulation of geodesic erosion presupposes the dual condition

$$f \geq g,$$

so that the mask acts as a lower bound throughout the entire process.

4.3.3.4 Implementation of geodesic erosion

The static function `mm::cero` implements this operator:

```
staticmethod
def cero(f, g, b=np.zeros((3,3),dtype='uint8'), n=1):
    """Erosão geodésica do marcador f sob a máscara g."""
    y = f.copy()
    for _ in range(n):
        y = np.maximum(mm.ero(y, b), g)
    return y
```

The instruction `np.maximum(mm::ero(y, b), g)` directly implements the expression $(y \ominus b) \vee g$.

As with geodesic dilation, the parameter n defines how many successive geodesic erosions will be computed before returning the final image.

> **i** Relation to morphological reconstruction
>
> The morphological reconstruction presented in the next section is obtained by iteratively applying geodesic dilation (`mm::cdil`) until a fixed point is reached, that is, until no cell changes value between two consecutive iterations. In other words, reconstruction consists of a sequence of successive geodesic dilations that propagate within the mask until no further changes occur.
> Dually, it is also possible to define reconstructions based on geodesic erosion through successive applications of `mm::cero`.

4.3.3.5 Example: propagation in a maze via duality

Figure 4.8 illustrates the resolution of the connectivity problem in a maze using morphological duality through the functions `mm::cero` and `mm::suprec`.

Instead of propagating a marker through the free corridors using geodesic dilations, the problem is formulated in the complementary domain. Initially, the original mask is inverted,

$$g = 1 - g_{orig},$$

so that the walls now assume value 1 and the corridors value 0. Similarly, the marker is constructed in this same complementary domain, containing a single value 0 at the maze entrance position and value 1 in all other pixels.

Using the cross-shaped structuring element (`mm::secross()`), the geodesic erosion acts on the complemented marker. At each iteration of `mm::cero`, the region connected to the initial marker undergoes successive erosions, while the mask imposes a lower limit that prevents propagation through the maze walls.

The intermediate images show evolution states after different numbers of iterations (`n=5`, `n=12`, and `n=22`). The final result is obtained by the geodesic reconstruction by erosion (`mm::suprec`), which applies successive geodesic erosions until reaching a fixed point, that is, a situation in which no further change occurs between two consecutive iterations.

In the complementary domain, the reconstructed region corresponds exactly to the set of corridors connected to the maze entrance. Thus, connectivity between entrance and exit can be determined directly from the reconstructed image.

```
%%writefile tmp/fig_04_cero_labirinto.cpp
#define MM_OUT "tmp/fig_04_cero_labirinto.png"
// Compile with: g++ -std=c++17 -o program program.cpp -I. -lmm -lpng -ljpeg -ltiff
#include "morph.hpp"
#include <vector>
#include <string>
#include <filesystem>

int main() {
```

```
    //| label: fig-04-cero-labirinto
    //| fig-cap: "Resolução de labirinto no domínio complementar: com máscara e marcador
    invertidos, *mm::cero* e *mm::suprec* propagam a onda geodésica pelos corredores."
    //| echo: true
    //| output: true

    // Máscara já invertida (255 = parede, 0 = corredor)
    mm::Image g(10, 10, 1);

    // Matrix definition
    unsigned char g_data[10][10] = {
        {255,  0,255,255,255,255,255,255,255,255},
        {255,  0,  0,  0,  0,  0,255,  0,  0,  0},
        {255,255,255,255,255,  0,255,  0,255,  0},
        {255,  0,  0,  0,255,  0,  0,  0,255,  0},
        {255,  0,255,  0,255,255,255,255,255,  0},
        {255,  0,255,  0,  0,  0,  0,  0,  0,  0},
        {255,  0,255,255,255,255,255,255,  0,255},
        {255,  0,  0,  0,  0,  0,  0,255,  0,255},
        {255,255,255,255,255,255,  0,  0,  0,255},
        {255,255,255,255,255,255,255,255,  0,255}
    };

    for (int y = 0; y < 10; y++) {
        for (int x = 0; x < 10; x++) {
            g.at(y, x) = g_data[y][x];
        }
    }

    mm::Image f(10, 10, 1);
    std::fill(f.data.begin(), f.data.end(), 255);
    f.at(0, 1) = 0;                       // semente (0) no corredor de entrada
    mm::Image B_cruz = mm::secross();

    mm::Image passo_5    = mm::cero(f, g, B_cruz, 5);
    mm::Image passo_12   = mm::cero(f, g, B_cruz, 12);
    mm::Image passo_22   = mm::cero(f, g, B_cruz, 22);
    mm::Image ponto_fixo = mm::suprec(f, g, B_cruz);

    mm::show(std::vector<mm::Image>{g, f, passo_5, passo_12, passo_22, ponto_fixo},
             MM_OUT,
             std::vector<std::string>{"Mascara (~g)", "Marcador (~f)", "n=5", "n=12", "n=22",
"~mm.suprec"},
             6);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(g, "tmp/fig_04_cero_labirinto_0.png");
mm::write(f, "tmp/fig_04_cero_labirinto_1.png");
mm::write(passo_5, "tmp/fig_04_cero_labirinto_2.png");
mm::write(passo_12, "tmp/fig_04_cero_labirinto_3.png");
mm::write(passo_22, "tmp/fig_04_cero_labirinto_4.png");
mm::write(ponto_fixo, "tmp/fig_04_cero_labirinto_5.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_04_cero_labirinto.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_cero_labirinto.cpp -o
tmp/fig_04_cero_labirinto -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_cero_labirinto \
  && test -f "tmp/fig_04_cero_labirinto.png" \
  || echo "  mm::show não gravou tmp/fig_04_cero_labirinto.png"
```

```
[1] Mascara (~g)
[2] Marcador (~f)
[3] n=5
[4] n=12
[5] n=22
[6] ~mm.suprec
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_04_cero_labirinto_0.png"),
            mm.read("tmp/fig_04_cero_labirinto_1.png"),
            mm.read("tmp/fig_04_cero_labirinto_2.png"),
            mm.read("tmp/fig_04_cero_labirinto_3.png"),
            mm.read("tmp/fig_04_cero_labirinto_4.png"),
            mm.read("tmp/fig_04_cero_labirinto_5.png"),
        ],
        titles=[
            'Mascara (~g)',
            'Marcador (~f)',
            'n=5',
            'n=12',
            'n=22',
            '~mm.suprec',
        ],
        cols=6,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_cero_labirinto_0.png (ver a versao Python)")
```

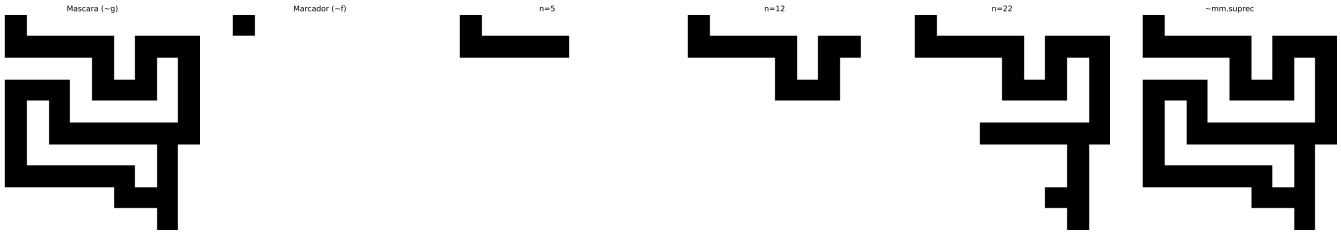


Figure 4.8: Resolução de labirinto no domínio complementar: com máscara e marcador invertidos, *mm::cero* e *mm::suprec* propagam a onda geodésica pelos corredores.

4.3.4 Morphological Reconstruction

Morphological reconstruction propagates a **marker** image f within a **mask** image g, ensuring that the result never exceeds the intensity values imposed by the mask. The fundamental operator that enables this contained propagation is the **geodesic dilation**, defined by Equation 4.7.

Reconstruction is obtained by the iterative application of this conditioned dilation. Initially, the marker is bounded by the mask to establish the initial state:

$$X^{(0)} = f \wedge g,$$

and the subsequent iterations are defined recursively by:

$$X^{(k)} = (X^{(k-1)} \oplus b) \wedge g.$$

The sequence grows monotonically until it reaches a fixed point, producing the **morphological reconstruction by dilation** (also known in the literature as *inf-reconstruction*):

$$R_g^{\delta}(f) = \lim_{k \to \infty} X^{(k)} = X^{(k)} \quad \text{when} \quad X^{(k)} = X^{(k-1)}. \tag{4.9}$$

The iterative ascent is halted as soon as stability is achieved, that is, when two consecutive iterations produce matrices with absolutely identical values.

4.3.4.1 Implementation of morphological reconstruction

The didactic routine `mm::infrec` directly implements the iterative fixed-point algorithm. Initially, the initial effective marker $X^{(0)}$ is determined by the operation `np.minimum(f, g)`. To ensure that the verification loop executes the first pass without triggering false premature convergences, the control variable of the previous iteration (`y1`) is initialized filled with a sentinel value outside the data domain (or simply with a matrix that forces the first execution).

```
def infrec(f, g, b=np.zeros((3,3), dtype='uint8')):
    """Inf-reconstruction: dilates the marker (f   g) until convergence under the mask g."""
    y = np.minimum(f, g)
    # Initialize y1 with impossible values to force entry into the loop
    y1 = np.full_like(f, 256, dtype=np.int16)
    while not np.array_equal(y, y1):
        y1 = y.copy()
        # Apply geodesic dilation: (y   b)   g
        y = np.minimum(mm.dil(y, b), g)
    return y.astype('uint8')
```

Inside the `while` loop, the variable `y` stores the current estimate of the reconstruction $X^{(k)}$, while `y1` preserves the image from the immediately preceding stage $X^{(k-1)}$. The conditional control statement `np.minimum(mm::dil(y, b), g)` faithfully translates the theoretical geodesic dilation, where the conventional morphological expansion commanded by OpenCV is immediately "pruned" and limited by the intensity barriers of the mask g. The loop ceases when no pixel modification is recorded between steps.

4.3.4.2 Advantages of Morphological Reconstruction

Morphological reconstruction is significantly more robust than conventional opening because it can remove unwanted structures without distorting or altering the morphology of the objects that should be preserved.

While classical opening smooths corners, eliminates tips, and deforms contours due to the rigid geometric imposition of the structuring element, geodesic reconstruction uses the mask to accurately recover the original boundaries and shapes of objects that have connectivity with the original marker.

In intuitive terms, the marker acts as a contagion seed that expands progressively, but only travels through the regions allowed by the mask. Components that have no intersection with the marker will never be reconstructed (being eliminated), while components touched by the seed expand until fully restoring their original geometry.

This discriminatory and conservative behavior is illustrated in Figure 4.9, using the cross-shaped structuring element presented in Figure 4.3.

```
%%writefile tmp/fig_04_reconstrucao_didatica.cpp
#define MM_OUT "tmp/fig_04_reconstrucao_didatica.png"
// Compile with: g++ -std=c++17 -o program program.cpp -I. -lstdc++ -lm
```

```cpp
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>

//| label: fig-04-reconstrucao-didatica
//| fig-cap: "*Pipeline* de Reconstrução Morfológica por Dilatação Condicionada: a máscara contém dois objetos, o marcador isola apenas o núcleo do objeto principal, e as iterações reconstroem sua forma exata até a convergência."
//| echo: true
//| output: true

int main() {
    mm::Image g(10, 10);
    unsigned char g_data[10][10] = {
        {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}};
    for (int y = 0; y < 10; y++)
        for (int x = 0; x < 10; x++)
            g.at(y, x) = g_data[y][x] * 255;

    mm::Image B_cruz = mm::secross();
    mm::Image f = mm::ero(g, mm::sebox());  // marcador: núcleo do objeto principal

    std::vector<mm::Image> iteracoes;
    std::vector<std::string> titulos;
    mm::Image img_atual = f;
    for (int i = 1; i <= 5; i++) {
        img_atual = mm::cdil(img_atual, g, B_cruz);
        iteracoes.push_back(img_atual);
        titulos.push_back("Dilatação Cond. (n=" + std::to_string(i) + ")");
    }

    mm::Image img_reconstruida = mm::infrec(f, g, B_cruz);

    // Check if reconstruction equals last iteration
    bool equal = true;
    for (int i = 0; i < img_reconstruida.h * img_reconstruida.w * img_reconstruida.channels;
    i++) {
        if (img_reconstruida.data[i] != iteracoes.back().data[i]) {
            equal = false;
            break;
        }
    }
    std::cout << "  Estabilidade na iteração 5: " << (equal ? "True" : "False") << std::endl;

    std::vector<mm::Image> imgs;
    imgs.push_back(g);
    imgs.push_back(f);
    imgs.insert(imgs.end(), iteracoes.begin(), iteracoes.end());
    imgs.push_back(img_reconstruida);
```

```
    std::vector<std::string> titles;
    titles.push_back("Máscara (g)");
    titles.push_back("Marcador (f)");
    titles.insert(titles.end(), titulos.begin(), titulos.end());
    titles.push_back("Reconstrução R_g(f)");

    mm::show(imgs, MM_OUT, titles, 8);

    return 0;
}
```

```
Overwriting tmp/fig_04_reconstrucao_didatica.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4
tmp/fig_04_reconstrucao_didatica.cpp -o tmp/fig_04_reconstrucao_didatica -lopencv_imgproc
-lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_reconstrucao_didatica \
  && test -f "tmp/fig_04_reconstrucao_didatica.png" \
  || echo "  mm::show não gravou tmp/fig_04_reconstrucao_didatica.png"
```

```
 Estabilidade na iteração 5: True
[1] Máscara (g)
[2] Marcador (f)
[3] Dilatação Cond. (n=1)
[4] Dilatação Cond. (n=2)
[5] Dilatação Cond. (n=3)
[6] Dilatação Cond. (n=4)
[7] Dilatação Cond. (n=5)
[8] Reconstrução R_g(f)
```

```
try:
    mm.show(mm.read("tmp/fig_04_reconstrucao_didatica.png"), figsize=(18, 3))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_reconstrucao_didatica.png (ver a versao Python)")
```

Figure 4.9: *Pipeline* de Reconstrução Morfológica por Dilatação Condicionada: a máscara contém dois objetos, o marcador isola apenas o núcleo do objeto principal, e as iterações reconstroem sua forma exata até a convergência.

4.3.5 Hole Filling and Border Removal

Two operators based on morphological reconstruction complete the binary cleaning *pipeline*. Their main characteristics are summarized in Table 4.3.

Hole filling (`mm::clohole`) removes cavities completely surrounded by the object, regardless of size, without altering the external contours. The procedure operates on the complement of the image, using as a marker a restriction of the frame to the background:

$$\text{clohole}(f) = \left(R^{\delta}_{f^c}(\text{frame}(f) \wedge f^c)\right)^c \tag{4.10}$$

In operational terms, the external background is first reconstructed, and then complementation is applied to recover the objects with filled holes.

Border object removal (`mm::edgeoff`) eliminates all objects that touch the image border, preserving only fully internal components. The marker is obtained by the intersection between the frame and the objects of the image:

$$\text{edgeoff}(f) = f \setminus R_f^{\delta}(\text{frame}(f) \wedge f) \tag{4.11}$$

Table 4.3: Comparison between the *clohole* and *edgeoff* operators.

Operator	Marker	Mask	Effect
`mm::clohole`	*frame* restricted to the background (f^c)	f^c	Fills internal holes
`mm::edgeoff`	*frame* restricted to the object (f)	f	Removes objects connected to the border

The step-by-step evolution of these geodesic transformations can be followed in the figures below. Figure 4.10 illustrates the controlled flooding mechanism of the `mm::clohole` operator, in which reconstruction occurs from the external background and prevents propagation into internal regions not connected to the exterior, resulting in consistent filling of internal cavities. In contrast, Figure 4.11 details the dynamics of the `mm::edgeoff` operator, in which only components connected to the border are reconstructed and subsequently removed, preserving exclusively the objects fully contained within the interior of the image.

The connectivity of the geodesic propagation is controlled by the structuring element: `mm::sebox()` (8-neighborhood) includes diagonal connections, whereas `mm::secross()` (4-neighborhood) excludes them. Consequently, the choice of the structuring element affects which components are reached by the reconstruction and, therefore, which will be preserved or removed.

4.3.5.1 Compliance with the implementation

The definitions above are directly aligned with the implementation in `morph.py`, reproduced below:

```
staticmethod
def clohole(f, b=np.ones((3,3),dtype='uint8')):
    # marcador restrito ao fundo da imagem
    marcador = mm.frame(f, border=1) & mm.neg(f)
    return mm.neg(mm.infrec(marcador, mm.neg(f), b))

staticmethod
def edgeoff(f, b=np.ones((3,3),dtype='uint8')):
    # marcador restrito aos objetos da imagem
    marcador = mm.frame(f, border=1) & f
    return mm.subm(f, mm.infrec(marcador, f, b))
```

These implementations make it explicit that both operators are direct instances of morphological reconstruction by geodesic dilation using `mm::infrec`, differing only in the choice of marker and mask: `clohole` operates on the image complement, while `edgeoff` operates directly in the domain of objects.

```
%%writefile tmp/fig_04_clohole_didatico.cpp
#define MM_OUT "tmp/fig_04_clohole_didatico.png"
//| label: fig-04-clohole-didatico
//| fig-cap: "*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."
```

```
//| echo: true
//| output: true

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>

int main() {
    mm::Image f(10, 10);
    std::vector<std::vector<int>> f_data = {
        {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}
    };
    for (int y = 0; y < 10; y++)
        for (int x = 0; x < 10; x++)
            f.at(y, x) = f_data[y][x] * 255;

    mm::Image B_cruz = mm::secross();
    mm::Image f_c = mm::neg(f);
    mm::Image marcador_ch = mm::frame(f, 1);       // borda externa

    std::vector<mm::Image> iteracoes;
    std::vector<std::string> titulos;
    mm::Image img_atual = marcador_ch;
    for (int i = 1; i <= 5; i++) {
        img_atual = mm::cdil(img_atual, f_c, B_cruz);
        iteracoes.push_back(img_atual);
        titulos.push_back("Iter. (n=" + std::to_string(i) + ")");
    }

    mm::Image img_clohole = mm::neg(mm::infrec(marcador_ch, f_c, B_cruz));
    mm::Image clohole_ref = mm::clohole(f);
    bool valid = true;
    for (int i = 0; i < img_clohole.h * img_clohole.w * img_clohole.channels; i++) {
        if (img_clohole.data[i] != clohole_ref.data[i]) {
            valid = false;
            break;
        }
    }
    std::cout << "  Validação clohole: " << (valid ? "True" : "False") << "\n";

    std::vector<mm::Image> all_images;
    all_images.push_back(f);
    all_images.push_back(marcador_ch);
    all_images.insert(all_images.end(), iteracoes.begin(), iteracoes.end());
    all_images.push_back(img_clohole);

    std::vector<std::string> all_titles;
    all_titles.push_back("f original");
    all_titles.push_back("Marcador (borda)");
    all_titles.insert(all_titles.end(), titulos.begin(), titulos.end());
    all_titles.push_back("clohole(f)");
```

```
    mm::show(all_images, MM_OUT, all_titles, 8);
    return 0;
}
```

```
Overwriting tmp/fig_04_clohole_didatico.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_clohole_didatico.cpp -o
tmp/fig_04_clohole_didatico -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_clohole_didatico \
  && test -f "tmp/fig_04_clohole_didatico.png" \
  || echo "  mm::show não gravou tmp/fig_04_clohole_didatico.png"
```

```
 Validação clohole: True
[1] f original
[2] Marcador (borda)
[3] Iter. (n=1)
[4] Iter. (n=2)
[5] Iter. (n=3)
[6] Iter. (n=4)
[7] Iter. (n=5)
[8] clohole(f)
```

```
try:
    mm.show(mm.read("tmp/fig_04_clohole_didatico.png"), figsize=(18, 3))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_clohole_didatico.png (ver a versao Python)")
```

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.

```
%%writefile tmp/fig_04_edgeoff_didatico.cpp
#define MM_OUT "tmp/fig_04_edgeoff_didatico.png"
// Compile with: g++ -std=c++17 -I. -o program program.cpp -lm

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>

int main() {
    //| label: fig-04-edgeoff-didatico
    //| fig-cap: "*Pipeline* of edge structure elimination (*edgeoff*): the marker captures
    the roots connected to the extremities, the reconstruction delimits these elements and the
    subtraction preserves only the totally internal objects."
    //| echo: true
    //| output: true

    mm::Image f(10, 10);
```

```
    // Initialize f with the given pattern * 255
    unsigned char pattern[10][10] = {
        {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}
    };
    for (int y = 0; y < 10; y++)
        for (int x = 0; x < 10; x++)
            f.at(y, x) = pattern[y][x] * 255;

    mm::Image B_box = mm::sebox();
    mm::Image marcador_eo = mm::band(mm::frame(f, 1), f);   // borda   f

    std::vector<mm::Image> iteracoes;
    std::vector<std::string> titulos;
    mm::Image img_atual = marcador_eo;
    for (int i = 1; i < 6; i++) {
        img_atual = mm::cdil(img_atual, f, B_box);
        iteracoes.push_back(img_atual);
        titulos.push_back("Iter. (n=" + std::to_string(i) + ")");
    }

    mm::Image img_edgeoff = mm::edgeoff(f, B_box);

    std::vector<mm::Image> show_imgs = {f, marcador_eo};
    show_imgs.insert(show_imgs.end(), iteracoes.begin(), iteracoes.end());
    show_imgs.push_back(img_edgeoff);

    std::vector<std::string> show_titles = {"f original", "Marcador (borda)"};
    show_titles.insert(show_titles.end(), titulos.begin(), titulos.end());
    show_titles.push_back("edgeoff(f)");

    mm::show(show_imgs, MM_OUT, show_titles, 8);

    return 0;
}
```

```
Overwriting tmp/fig_04_edgeoff_didatico.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_edgeoff_didatico.cpp -o
tmp/fig_04_edgeoff_didatico -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_edgeoff_didatico \
  && test -f "tmp/fig_04_edgeoff_didatico.png" \
  || echo "  mm::show não gravou tmp/fig_04_edgeoff_didatico.png"
```

```
[1] f original
[2] Marcador (borda)
[3] Iter. (n=1)
[4] Iter. (n=2)
[5] Iter. (n=3)
[6] Iter. (n=4)
[7] Iter. (n=5)
[8] edgeoff(f)
```

```
try:
    mm.show(mm.read("tmp/fig_04_edgeoff_didatico.png"), figsize=(18, 3))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_edgeoff_didatico.png (ver a versao Python)")
```

Figure 4.11: *Pipeline* de eliminação de estruturas de borda (*edgeoff*): o marcador captura as raízes conectadas às extremidades, a reconstrução delimita esses elementos e a subtração preserva só os objetos totalmente internos.

4.3.6 Binary Cleaning Pipeline with CLAHE

Based on the previous analysis — in which CLAHE produced the highest inter-class variance value ($\sigma_B^2 \approx 2.47 \times 10^3$), see Figure 4.2, and the `mm::clohole` operator proved effective in filling internal cavities —, the final segmentation pipeline, illustrated in Figure 4.12, is structured by the following computational flow:

$$\text{gray} \xrightarrow{\text{CLAHE}} \xrightarrow{\text{Otsu}} \xrightarrow{\text{open}} \xrightarrow{\text{clohole}} \xrightarrow{\text{open}} \xrightarrow{\text{edgeoff}} \text{segmentation}$$

After the `mm::clohole` step, a second morphological opening is applied with a larger structuring element (`mm::sedisk(33)`, a disk of diameter 33). This operation removes small residual regions and artifacts that may have remained after segmentation. In particular, geodesic filling can transform small isolated cavities into components connected to the object, making an additional size-based filtering step convenient. The diameter was chosen so that the coins remain able to contain the structuring element, while significantly smaller components are eliminated.

In this image, no coin is connected to the matrix border. Consequently, applying `mm::edgeoff` does not alter the result obtained after the second opening. Nevertheless, this step is kept in the pipeline for robustness, since in other images there may be partially visible objects or objects connected to the borders, which should be removed before the analysis stage.

Why the opening after clohole?

The `mm::clohole` operator fills all closed cavities present in the segmented objects. In some situations, small unwanted regions may remain after this step or become connected to the main objects. The subsequent morphological opening removes components smaller than the structuring element, while preserving the coins due to their significantly larger size.

```
%%writefile tmp/fig_04_pipeline_clahe.cpp
#define MM_OUT "tmp/fig_04_pipeline_clahe.png"
// Compile with: g++ -std=c++17 -o program program.cpp -I. -lmm -lopencv_core -lopencv_imgproc
-lopencv_highgui

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <filesystem>

int main() {
```

```
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_coins_gray = mm::_read_state("tmp/state/img_coins_gray_8.png");
// [pdi:state-io:end]

    //   Pre-processing: only CLAHE
    mm::Image img_clahe0 = mm::clahe(img_coins_gray, 2.0, 8);

    //   Step 1: Otsu binarization
    mm::Image img_bin = mm::threshold(img_clahe0);

    //   Step 2: Opening - removes white background noise
    mm::Image img_open = mm::open(img_bin, mm::sedisk(9));

    //   Step 3: clohole - closes all internal holes
    mm::Image img_hole = mm::clohole(img_open);

    //   Step 4: Opening (large kernel) - removes clohole artifacts
    mm::Image img_limpo = mm::open(img_hole, mm::sedisk(33));

    //   Step 5: edgeoff - removes objects touching the border
    mm::Image img_final = mm::edgeoff(img_limpo, mm::SE::box(3), 1);

    mm::show(
        std::vector<mm::Image>{img_clahe0, img_bin, img_open,
                               img_hole, img_limpo, img_final},
        MM_OUT,
        std::vector<std::string>{"CLAHE", "Otsu", "Abertura (r=9)",
                                 "clohole", "Abertura (r=33)", "Final (edgeoff)"},
        6
    );


// [pdi:state-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp/state");
mm::write(img_final, "tmp/state/img_final_55.png");
// [pdi:state-io:end]

// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_clahe0, "tmp/fig_04_pipeline_clahe_0.png");
mm::write(img_bin, "tmp/fig_04_pipeline_clahe_1.png");
mm::write(img_open, "tmp/fig_04_pipeline_clahe_2.png");
mm::write(img_hole, "tmp/fig_04_pipeline_clahe_3.png");
mm::write(img_limpo, "tmp/fig_04_pipeline_clahe_4.png");
mm::write(img_final, "tmp/fig_04_pipeline_clahe_5.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_04_pipeline_clahe.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_pipeline_clahe.cpp -o
tmp/fig_04_pipeline_clahe -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_pipeline_clahe \
  && test -f "tmp/fig_04_pipeline_clahe.png" \
  || echo "  mm::show não gravou tmp/fig_04_pipeline_clahe.png"
```

```
[1] CLAHE
[2] Otsu
[3] Abertura (r=9)
```

```
[4] clohole
[5] Abertura (r=33)
[6] Final (edgeoff)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_04_pipeline_clahe_0.png"),
            mm.read("tmp/fig_04_pipeline_clahe_1.png"),
            mm.read("tmp/fig_04_pipeline_clahe_2.png"),
            mm.read("tmp/fig_04_pipeline_clahe_3.png"),
            mm.read("tmp/fig_04_pipeline_clahe_4.png"),
            mm.read("tmp/fig_04_pipeline_clahe_5.png"),
        ],
        titles=[
            'CLAHE',
            'Otsu',
            'Abertura (r=9)',
            'clohole',
            'Abertura (r=33)',
            'Final (edgeoff)',
        ],
        cols=6,
        figsize=(18, 6),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_pipeline_clahe_0.png (ver a versao Python)")
```

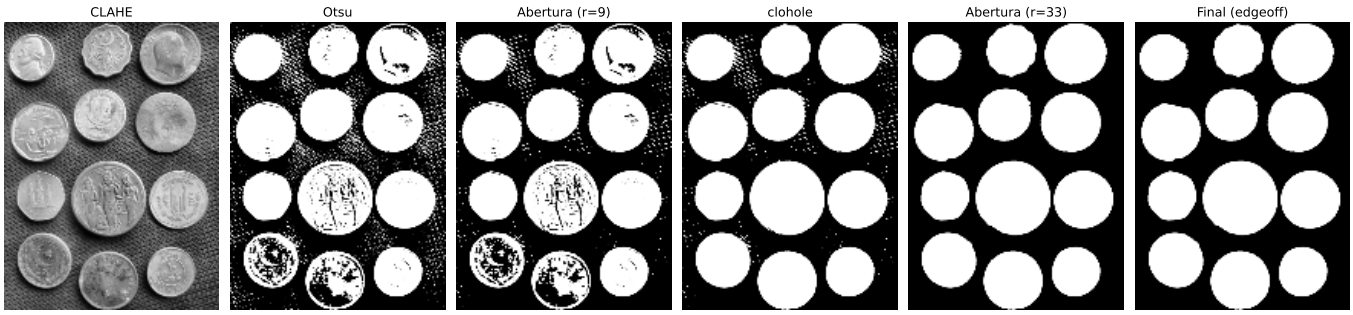


Figure 4.12: *Pipeline* completo de segmentação com CLAHE: Otsu → abertura (r=9) → clohole → abertura (r=33) → edgeoff.

4.3.7 Grayscale Morphology

Morphological operators extend naturally to grayscale images. In this formulation, erosion and dilation act directly on the image intensity levels. For flat structuring elements ($b \equiv 0$), erosion corresponds to the **local minimum** and dilation to the **local maximum** within the neighborhood defined by the structuring element.

The intuitive interpretation is straightforward: erosion **darkens** regions by replacing each pixel with the smallest value in its neighborhood, while dilation **lightens** regions by using the largest available value. Combining these operators allows for the construction of transformations capable of enhancing edges, removing illumination trends, and highlighting local structures.

Three derived operators are especially useful:

Morphological gradient — highlights edges as the difference between dilation and erosion:

$$\text{grad}_B(f) = (f \oplus B) - (f \ominus B) \tag{4.12}$$

Top-hat — highlights bright structures smaller than the structuring element (difference between the original image and its opening):

$$\text{top-hat}_B(f) = f - (f \circ B) \tag{4.13}$$

Black-hat — highlights dark structures smaller than the structuring element (difference between the closing and the original image):

$$\text{black-hat}_B(f) = (f \bullet B) - f \tag{4.14}$$

The *Top-hat* extracts bright details that do not survive the opening, while the *Black-hat* reveals dark details removed by the closing. The morphological gradient, in turn, enhances abrupt intensity transitions, producing a representation similar to that of an edge detector.

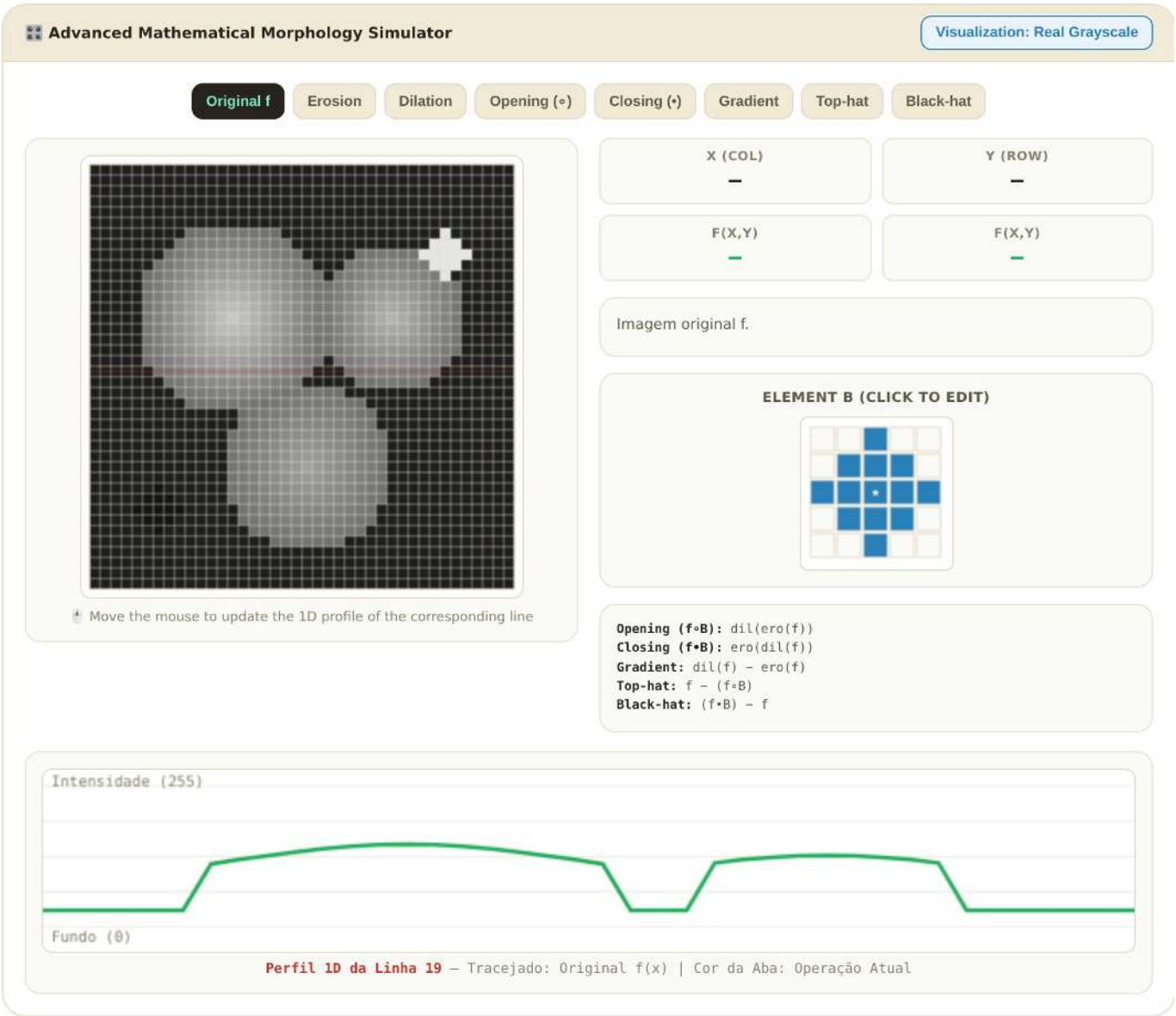
To understand the mechanism of these operators at a local level, Figure 4.13 presents an interactive simulator of grayscale morphology. The simulator allows you to freely edit the structuring element, visualize its displacement over the image, and simultaneously track the one-dimensional intensity profile. In this way, it becomes possible to directly observe how erosion selects local minima, how dilation selects local maxima, and how the morphological gradient emerges from the difference between these two operators.

The button located in the upper right corner allows you to toggle between the original grayscale visualization and a pseudocolored representation (*colormap*) for the three gradient types only. The colored version facilitates the visual perception of intensity variations, making the action of the morphological operators on maxima, minima, and local transitions of the image more evident.

The operators presented are available in `morph.py` through the functions `mm::gradm`, `mm::tophat`, and `mm::blackhat`.

Figure 4.14 illustrates the effects of these operators on the coin image and their respective histograms. Note that erosion shifts the distribution toward lower intensities, while dilation shifts it toward higher intensities. The gradient concentrates values in contour regions, and the *Top-hat* and *Black-hat* operators produce histograms strongly concentrated at low intensity levels, as only small local structures are highlighted.

```
%%writefile tmp/fig_04_morf_gc_histogramas.cpp
#define MM_OUT "tmp/fig_04_morf_gc_histogramas.png"
#include "morph.hpp"
#include <opencv2/opencv.hpp>
#include <iostream>
#include <vector>
#include <string>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_coins_gray = mm::_read_state("tmp/state/img_coins_gray_8.png");
// [pdi:state-io:end]

    //| label: fig-04-morf-gc-histogramas
    //| fig-cap: "Morfologia em tons de cinza e seus histogramas: erosão, dilatação,
    gradiente, top-hat e black-hat. Elemento estruturante: disco de diâmetro 19."
    //| echo: true
    //| output: true

    mm::Image B = mm::sedisk(19);
    mm::Image img_ero  = mm::ero(img_coins_gray, B);
    mm::Image img_dil  = mm::dil(img_coins_gray, B);
    mm::Image img_grad = mm::gradm(img_coins_gray, B);
    mm::Image img_th   = mm::tophat(img_coins_gray, B);
    mm::Image img_bh   = mm::blackhat(img_coins_gray, B);
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-04-operadores-av.html

Figure 4.13: Advanced interactive morphology simulator with editable structuring element and 1D profile.

```
    mm::show(
        std::vector<mm::Image>{img_coins_gray, mm::histImg(img_coins_gray), img_ero,
mm::histImg(img_ero),
         img_dil, mm::histImg(img_dil), img_grad, mm::histImg(img_grad),
         img_th, mm::histImg(img_th), img_bh, mm::histImg(img_bh)},
        MM_OUT,
        std::vector<std::string>{"Original", "Hist", "Erosão", "Hist", "Dilatação", "Hist",
                "Gradiente", "Hist", "Top-hat", "Hist", "Black-hat", "Hist"},
        4
    );

    return 0;
}
```

```
Overwriting tmp/fig_04_morf_gc_histogramas.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_morf_gc_histogramas.cpp
-o tmp/fig_04_morf_gc_histogramas -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_morf_gc_histogramas \
  && test -f "tmp/fig_04_morf_gc_histogramas.png" \
  || echo "  mm::show não gravou tmp/fig_04_morf_gc_histogramas.png"
```

```
[1] Original
[2] Hist
[3] Erosão
[4] Hist
[5] Dilatação
[6] Hist
[7] Gradiente
[8] Hist
[9] Top-hat
[10] Hist
[11] Black-hat
[12] Hist
```

```
try:
    mm.show(mm.read("tmp/fig_04_morf_gc_histogramas.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_morf_gc_histogramas.png (ver a versao Python)")
```

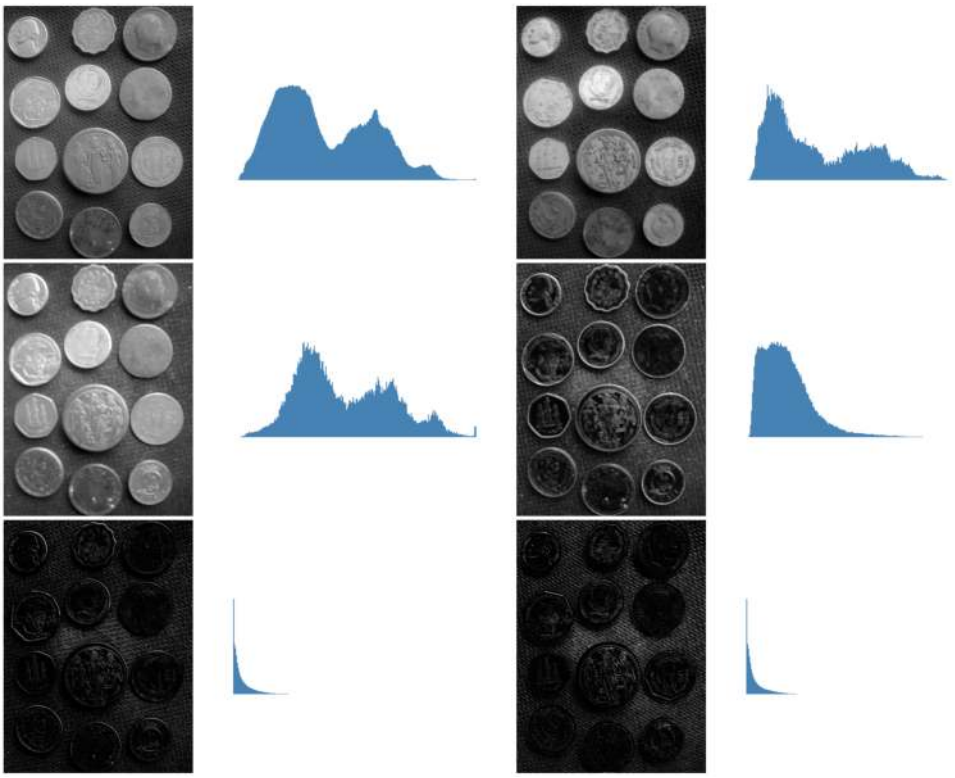


Figure 4.14: Morfologia em tons de cinza e seus histogramas: erosão, dilatação, gradiente, top-hat e black-hat. Elemento estruturante: disco de diâmetro 19.

The morphological operators presented previously will now be used as tools for refinement and marker generation in more advanced segmentation methods, presented below.

4.4 Image Segmentation: Fundamentals and Taxonomy

Image segmentation consists of dividing the image into regions associated with objects or structures of interest. In DIP, it represents the transition between low-level processing — such as filtering and enhancement — and more advanced analysis stages, such as feature extraction, recognition, and scene interpretation.

Formally, the goal of segmentation is to decompose the complete spatial domain of an image, denoted by Ω, into a partition of subsets $\{R_1, R_2, \ldots, R_n\}$ that simultaneously satisfies the criteria of **completeness** and **disjointness**:

$$\bigcup_{i=1}^{n} R_i = \Omega, \qquad R_i \cap R_j = \emptyset \quad \forall\, i \neq j \tag{4.15}$$

In addition to the completeness and disjointness properties expressed in Equation 4.15, each subregion R_i must constitute a **homogeneous** domain according to a similarity predicate defined over local properties — intensity, color, or texture — and, simultaneously, be **distinct** from adjacent regions.

Segmentation techniques can be organized into different families. This chapter emphasizes the approaches summarized in Table 4.4, based primarily on criteria of intensity, connectivity, and spatial proximity.

Table 4.4: Simplified taxonomy of the main segmentation and refinement approaches studied in this chapter.

Approach	Segmentation Criterion	Reference Operators
Thresholding	Partitioning of the intensity space	Otsu's criterion, global and local thresholding
Mathematical Morphology	Spatial relations defined by structuring elements/functions	Erosion, dilation, opening, closing, and reconstruction
Region-Based	Local homogeneity and spatial connectivity	Connected component labeling, Distance Transform, and *Watershed*

Up to this point, the practical development has focused on **thresholding**, through the combination of adaptive CLAHE equalization and Otsu's global method. This step was complemented by **morphological reconstruction** operators based on geodesic dilations, implemented by the functions `mm::infrec`, `mm::clohole`, and `mm::edgeoff`, producing a clean binary mask suitable for analysis.

However, in scenarios where distinct objects appear connected in the binary mask — whether by physical contact, partial overlap, or narrow pixel bridges produced by segmentation — thresholding is no longer sufficient to individualize each object. In such cases, multiple objects become part of a single connected component, hindering subsequent measurement and interpretation stages.

To overcome this limitation, the following sections introduce three complementary tools: **Connected Component Labeling**, the **Distance Transform**, and the marker-based ***Watershed*** segmentation algorithm. Together, these techniques make it possible to separate adjacent objects, identify regions individually, and extract consistent geometric descriptors for quantitative analysis.

4.4.1 Labeling

Connected component labeling is the operator that assigns a unique integer identifier to each set of pixels belonging to the same connected component in a binary image.

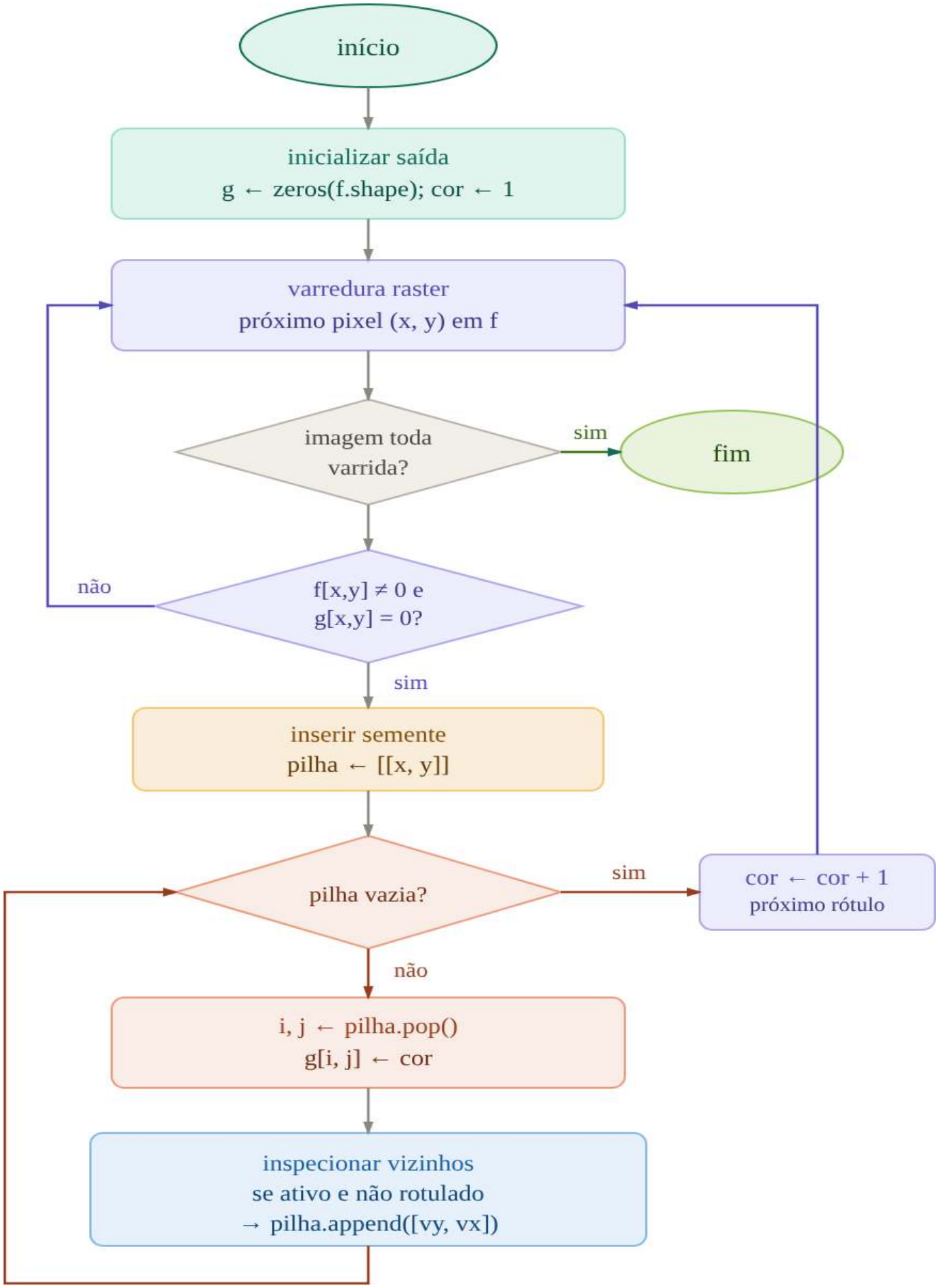
> **i** Formal Definition
>
> Given a binary image f and a connectivity relation defined by a structuring element B (typically 4-connectivity or 8-connectivity), the labeling algorithm of Figure 4.15 produces an image g in which all pixels belonging to the same connected component receive the same positive integer label, while pixels belonging to distinct components receive different labels.

Connectivity defines which pixels are considered direct neighbors of a pixel (x, y). The most commonly used definitions are:

- **4-Connectivity:** considers only the four orthogonal neighbors (north, south, east, and west).
- **8-Connectivity:** considers the four orthogonal neighbors and the four diagonal ones, totaling eight neighbors.

The choice of connectivity directly influences the formation of connected components and, consequently, the labeling result, as illustrated in Figure 4.17. An additional example can be interactively explored in the simulator presented in Figure 4.16.

The implementation in `morph.py` provides two versions of this operator. The function `mm::label0` explicitly reproduces the flood-fill algorithm using a stack and allows controlling connectivity through the adopted structuring element. Meanwhile, `mm::label` delegates the operation to OpenCV's optimized implementation (`mm::label0`). In both cases, the result is a labeled image in which each connected component receives a distinct integer identifier.

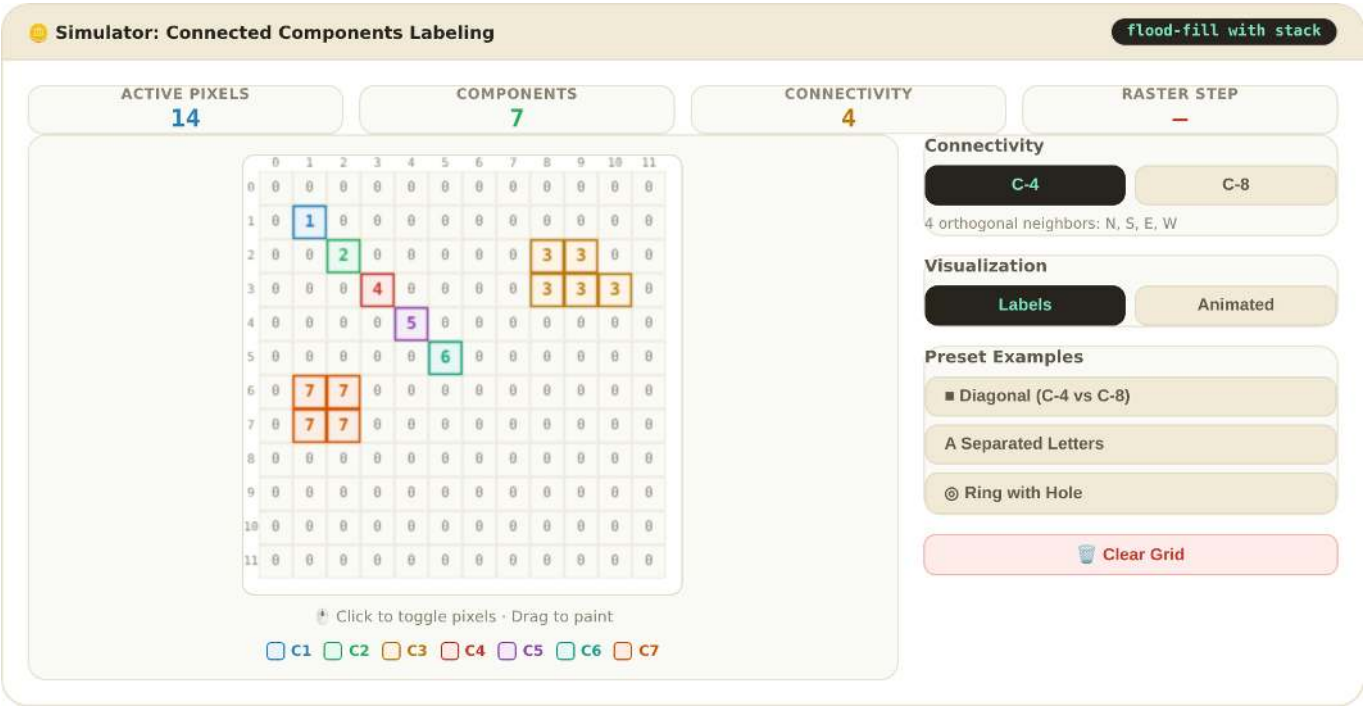


Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-alg-rotulagem2.html

Figure 4.15: Flood-fill labeling algorithm with stack.

The helper `_viz` iterates over the structuring window `b` centered at (i, j), generating only the valid neighbors within the image boundaries — the desired connectivity is entirely determined by the shape of `b` passed to the algorithm.

Didactic example — effect of connectivity:



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-04-rotulacao.html

Figure 4.16: Interactive simulator for connected component labeling: visualization of flood-fill expansion, 4-connectivity and 8-connectivity.

```
%%writefile tmp/fig_04_rotulacao_didatico.cpp
#define MM_OUT "tmp/fig_04_rotulacao_didatico.png"
//| label: fig-04-rotulacao-didatico
//| fig-cap: "Efeito da conectividade na rotulação (mm::label0): pixels diagonalmente
adjacentes formam componentes distintas em conectividade-4 e se fundem em conectividade-8."
//| echo: true
//| output: true

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

int main() {
    // Create the binary image f (10x10) from the given array
    mm::Image f(10, 10);
    unsigned char pattern[10][10] = {
        {0,0,0,0,0,0,0,0,0,0},
        {0,1,0,0,0,0,0,0,0,0},
        {0,0,1,0,0,0,0,0,0,0},
        {0,0,0,1,0,0,1,1,0,0},
        {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},
        {0,0,0,0,0,0,0,0,0,0}
```

```
    };
    for (int y = 0; y < 10; ++y) {
        for (int x = 0; x < 10; ++x) {
            f.at(y, x) = pattern[y][x] * 255;
        }
    }

    // B4 = mm.secross() - connectivity-4 structuring element
    // B8 = mm.sebox() - connectivity-8 structuring element
    mm::Image B4 = mm::secross();
    mm::Image B8 = mm::sebox();

    // Label connected components with 4- and 8-connectivity
    mm::Image lbl4 = mm::label0(f, B4);
    mm::Image lbl8 = mm::label0(f, B8);

    // Find the maximum label value in each
    int n4 = 0, n8 = 0;
    for (int i = 0; i < lbl4.h * lbl4.w; ++i) {
        n4 = std::max(n4, (int)lbl4.data[i]);
    }
    for (int i = 0; i < lbl8.h * lbl8.w; ++i) {
        n8 = std::max(n8, (int)lbl8.data[i]);
    }
    std::cout << "Componentes C4: " << n4 << "  |  C8: " << n8 << "\n";

    // Helper: normalize labels to [0,255] for display
    auto norm_label = [](const mm::Image& lbl, int mx) {
        mm::Image out = lbl;
        for (int y = 0; y < lbl.h; ++y) {
            for (int x = 0; x < lbl.w; ++x) {
                if (lbl.at(y, x)) {
                    out.at(y, x) = (int)(lbl.at(y, x) * 255 / mx);
                }
            }
        }
        return out;
    };

    // Display: original image, C4-labeled, C8-labeled
    mm::show(
        std::vector<mm::Image>{f, norm_label(lbl4, std::max(n4, 1)), norm_label(lbl8,
std::max(n8, 1))},
        MM_OUT,
        std::vector<std::string>{"f original", "C4 (" + std::to_string(n4) + " comp.)", "C8 ("
+ std::to_string(n8) + " comp.)"},
        3
    );

    return 0;
}
```

```
Overwriting tmp/fig_04_rotulacao_didatico.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_rotulacao_didatico.cpp
-o tmp/fig_04_rotulacao_didatico -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_rotulacao_didatico \
  && test -f "tmp/fig_04_rotulacao_didatico.png" \
  || echo "  mm::show não gravou tmp/fig_04_rotulacao_didatico.png"
```

```
Componentes C4: 7  |  C8: 4
[1] f original
[2] C4 (7 comp.)
[3] C8 (4 comp.)
```

```
try:
    mm.show(mm.read("tmp/fig_04_rotulacao_didatico.png"))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_rotulacao_didatico.png (ver a versao Python)")
```

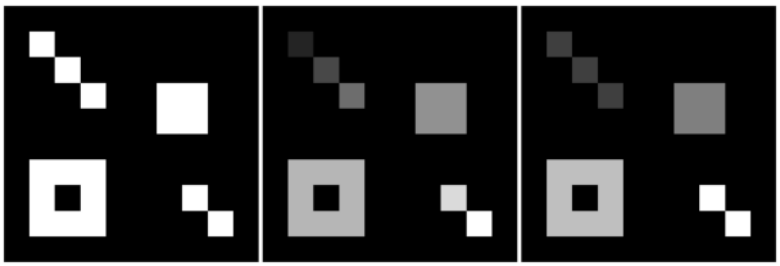


Figure 4.17: Efeito da conectividade na rotulação (*mm::label0*): pixels diagonalmente adjacentes formam componentes distintas em conectividade-4 e se fundem em conectividade-8.

4.4.2 Distance Transform

The **Distance Transform** (DT) is an operator that, applied to a binary image f, produces a grayscale image D in which each pixel belonging to the object ($f(x, y) \neq 0$) receives as its value the geometric distance to the nearest background pixel ($f(x', y') = 0$):

$$D(x, y) = \min_{(x', y') \,:\, f(x', y') = 0} d\big((x, y),\, (x', y')\big)$$

where $d(\cdot, \cdot)$ is a distance metric — typically the Euclidean distance (L_2). The result is a topographic representation of the objects: pixels located in the interior assume high values, whereas pixels near the borders exhibit low distance values. The **local maxima** of D correspond to the points farthest from the object boundary, often near their geometric centers or centers of maximal inscription — a property particularly useful for the automatic generation of markers in the *watershed* algorithm.

i Formal definition using erosions

The DT also admits an iterative morphological interpretation, according to the algorithm in Figure 4.18. Consider a structuring function b whose central value is zero and whose neighbors have negative costs associated with the displacement. By applying successive erosions with this particular structuring function, the pixel values of the objects (which must assume the maximum possible distance in the image) are progressively reduced according to the costs defined by b. The accumulated value of this propagation then comes to represent the distance to the background according to the metric induced by the structuring function.

This interpretation is implemented in `mm::dist1()`, which accumulates successive erosions using the operation `mm::ero1()`. In turn, `mm::dist()` delegates the Euclidean distance computation to the optimized OpenCV operator `mm::dist(f)`, where `f` is the input binary image, the L2 distance specifies the Euclidean metric (L_2), and `5` indicates the use of a 5×5 mask to approximate the distance with high precision.

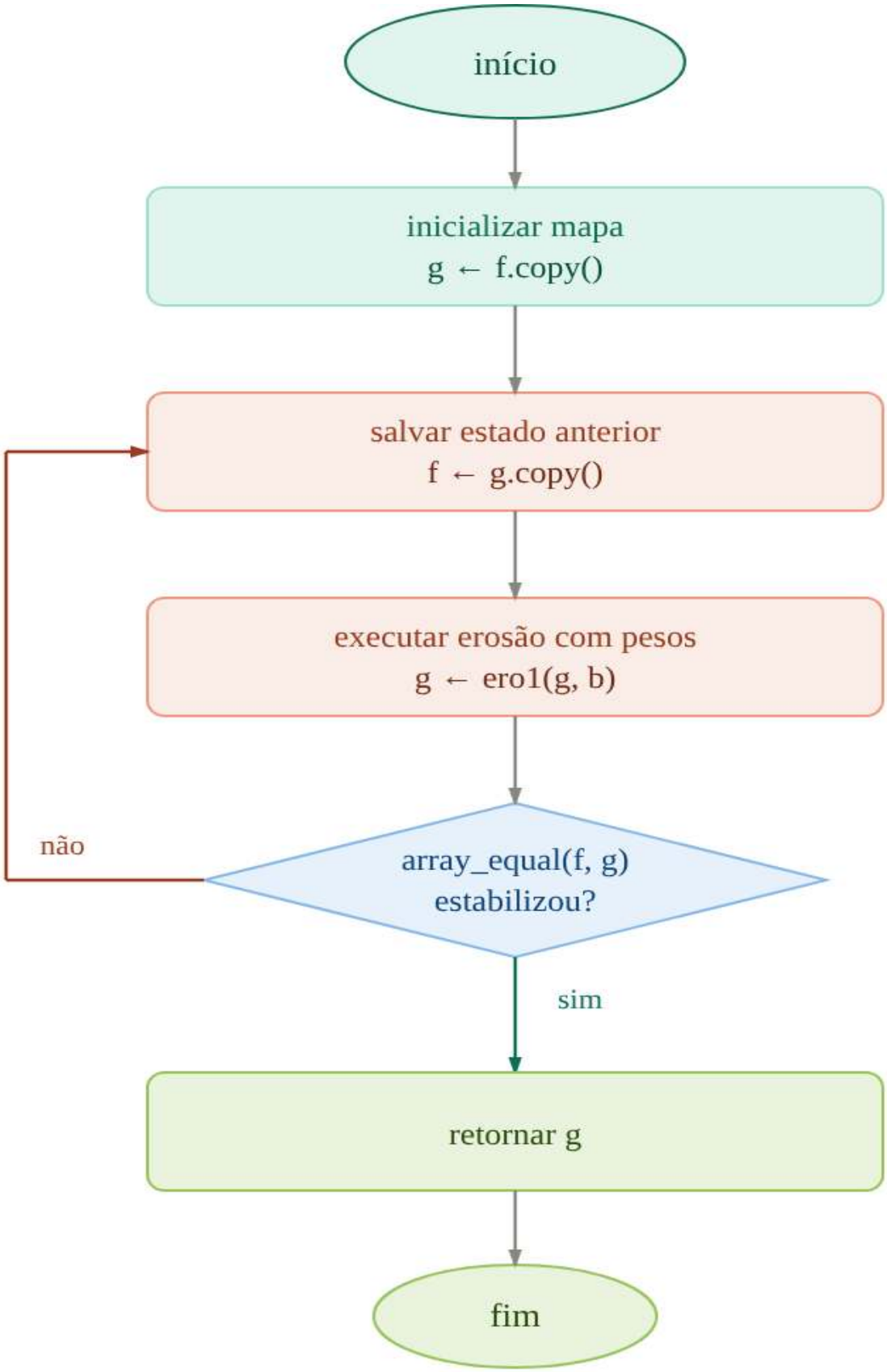
The `dist1` function produces a discrete distance transform whose metric is determined by the geometry and weights of the structuring function used. For example, using a cross-shaped structuring function with unit

cost for the four orthogonal neighbors yields the ***Manhattan*** distance (L_1). Other choices of neighborhood and weights induce different metrics. In turn, `mm::dist()` computes an efficient approximation of the Euclidean distance (L_2).

Because it requires successive erosions over the entire image, the `dist1` approach has a significantly higher computational cost than `mm::dist()`, and it is employed in this book mainly for didactic purposes and to highlight the relationship between mathematical morphology and distance transforms.

Figure 4.19 presents an interactive simulator for the TD: by positioning the cursor over different pixels of the object, it is possible to observe, in real time, the distance value associated with that position—that is, the distance to the nearest background pixel. Figure 4.20 presents a practical example of this execution in a Python environment.

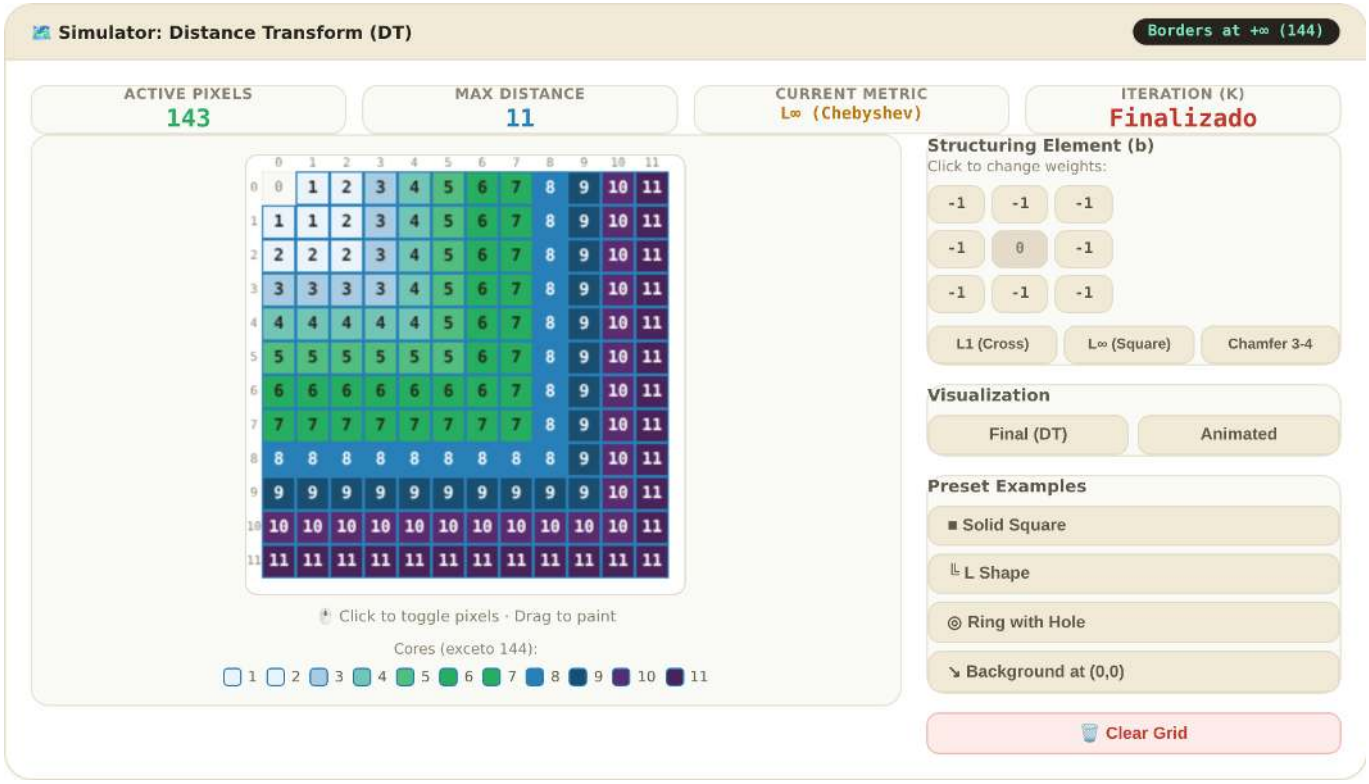
```
%%writefile tmp/fig_04_distancia_didatico.cpp
#define MM_OUT "tmp/fig_04_distancia_didatico.png"
//| label: fig-04-distancia-didatico
//| fig-cap: "Transformada de Distância em imagem binária 10×10. Esquerda: original
(*foreground* = 255). Centro: mm.dist1 iterativa (erosões com cruz). Direita: mm.dist (L2)."
//| echo: true
//| output: true

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <filesystem>

int main() {
    mm::Image f(10, 10);
    std::fill(f.data.begin(), f.data.end(), 255);
    f.at(0, 0) = 0;

    mm::SE B_cruz{{mm::SE_OUT, -1, mm::SE_OUT}, {-1, 0, -1}, {mm::SE_OUT, -1, mm::SE_OUT}};

    mm::Image d_iter = mm::dist1(f, B_cruz);
    mm::Image d_l2   = mm::dist(f);

    int max_d_iter = 0;
    int max_d_l2 = 0;
    for (int i = 0; i < d_iter.h * d_iter.w; ++i) {
        max_d_iter = std::max(max_d_iter, (int)d_iter.data[i]);
        max_d_l2 = std::max(max_d_l2, (int)d_l2.data[i]);
    }

    std::cout << "Máx. dist1 (erosões) : " << max_d_iter << " px\n";
    std::cout << "Máx. dist  (L2)      : " << max_d_l2 << " px\n";

    mm::show(
        std::vector<mm::Image>{f, d_iter, d_l2},
        MM_OUT,
        std::vector<std::string>{"f original", "mm.dist1 (erosões com cruz)", "mm.dist (L2)"},
        3
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(f, "tmp/fig_04_distancia_didatico_0.png");
mm::write(d_iter, "tmp/fig_04_distancia_didatico_1.png");
mm::write(d_l2, "tmp/fig_04_distancia_didatico_2.png");
// [pdi:panel-io:end]
```



Ponto Fixo Numérico: A distância emerge da propagação matemática do valor zero.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-alg-distancia2.html

Figure 4.18: Distance Transform Algorithm.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-04-distancia.html

Figure 4.19: Interactive simulator of the Distance Transform (DT) iterative via grayscale erosion. Pixels outside the image assume the maximum value (144), propagating costs from the internal background.

```
return 0;
}
```

```
Overwriting tmp/fig_04_distancia_didatico.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_distancia_didatico.cpp
-o tmp/fig_04_distancia_didatico -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_distancia_didatico \
  && test -f "tmp/fig_04_distancia_didatico.png" \
  || echo "  mm::show não gravou tmp/fig_04_distancia_didatico.png"
```

```
Máx. dist1 (erosões) : 18 px
Máx. dist  (L2)      : 13 px
[1] f original
[2] mm.dist1 (erosões com cruz)
[3] mm.dist (L2)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_04_distancia_didatico_0.png"),
            mm.read("tmp/fig_04_distancia_didatico_1.png"),
            mm.read("tmp/fig_04_distancia_didatico_2.png"),
        ],
        titles=[
            'f original',
            'mm.dist1 (erosões com cruz)',
            'mm.dist (L2)',
        ],
```

```
        cols=3,
        axis=True,
        figsize=(12, 4),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_distancia_didatico_0.png (ver a versao Python)")
```

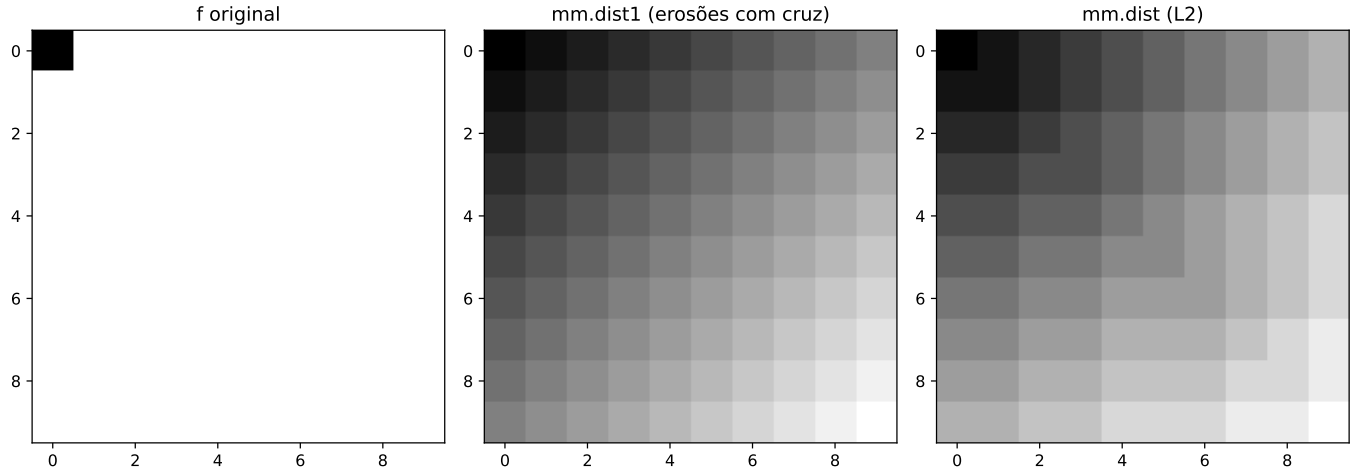


Figure 4.20: Transformada de Distância em imagem binária 10×10. Esquerda: original (*foreground* = 255). Centro: mm.dist1 iterativa (erosões com cruz). Direita: mm.dist (L2).

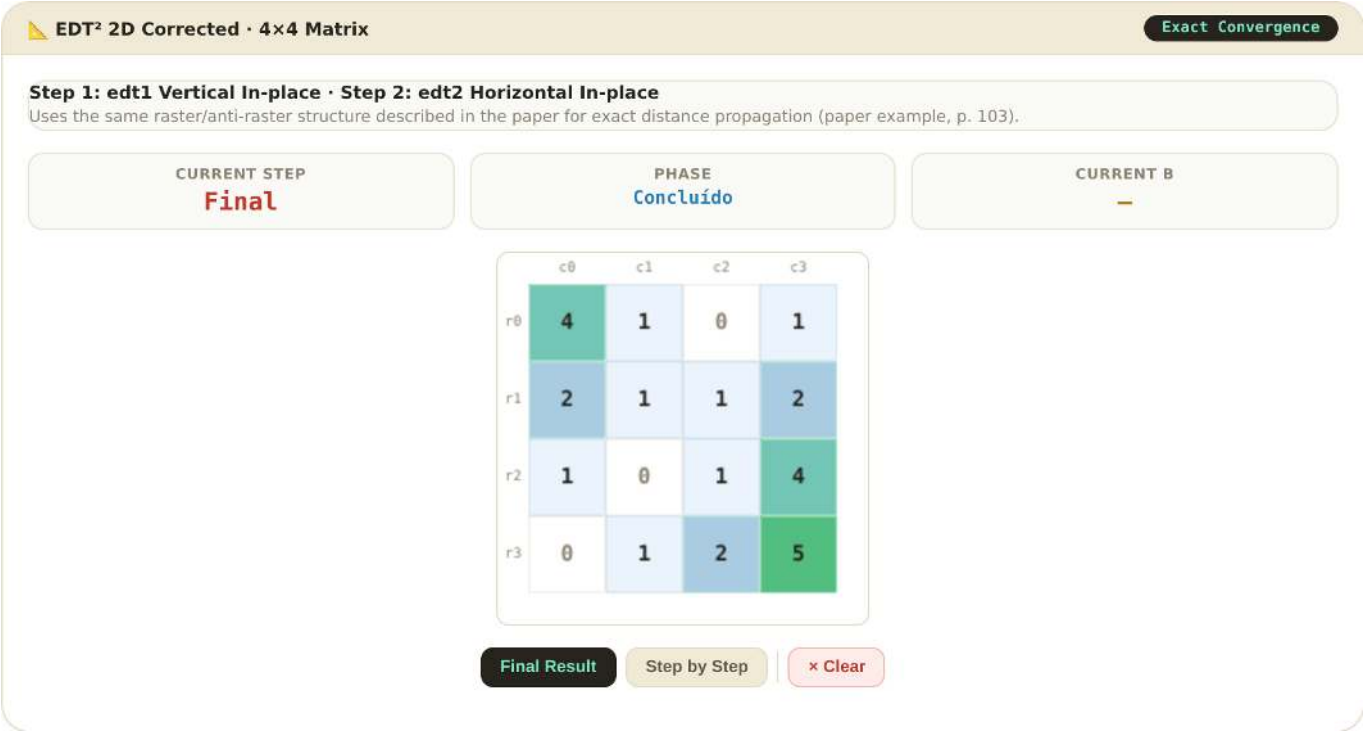
Annotating numerical values directly over the pixels allows verifying how `dist1` propagates distances according to the metric induced by the structuring function used. In the case of the cross element with unit cost, the obtained values correspond to the *Manhattan* distance (L_1). Although `dist1` and `mm::dist` produce distinct numerical values because they adopt different metrics, both transforms preserve the topographical structure of the objects, causing their maxima to occur in similar central regions. This property justifies the use of `mm::dist` in practical applications, due to its high computational efficiency.

4.4.3 Euclidean Distance Transform in four steps

The simulator in Figure 4.21 implements the EDT algorithm from Lotufo (2001) in two stages. In the first stage, the function `edt1` performs a one-dimensional vertical transformation sequentially (*in-place*), traversing each column in *raster* (↓) and *anti-raster* (↑) order to compute distances in the vertical direction (two steps: South and North). In the second stage, the function `edt2` uses this result as input and performs horizontal propagation using queues: for each row of the matrix, two priority queues, `Eq` and `Wq`, are initialized by traversing the column indices in opposite directions (`Eq` from `W-1` to `1`, `Wq` from `2` to `W`), so that each updated pixel immediately enqueues its neighbors for reprocessing within the same round (two more steps: East and West). This queue mechanism allows successive erosions with increasing odd weights (`b = 1, 3, 5, ...`, incremented at each iteration of the outer loop) without propagation getting stuck on outdated values, since each round resolves the horizontal dependency chain completely before the next increment of `b`. The convergence of this propagation produces the Euclidean Distance Transform across the entire matrix, combining the vertical information obtained in `edt1` with the queue-based horizontal propagation performed in `edt2`.

4.4.3.1 Geodesic Distance Transform

The geodesic distance transform associates each pixel with the smallest distance to a marker, under the constraint imposed by a mask. Thus, propagation occurs exclusively through allowed pixels, preserving the connectivity of the domain.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-04-tde.html

Figure 4.21: Interactive 2D simulator (4x4) with strict synchronization of horizontal propagation queues (b) to obtain the exact convergence described in the paper.

Figure 4.22 illustrates this process in a maze: (a) the mask `g`; (b) the geodesic distance `D1` calculated from the entrance; (c) the distance `D2` calculated from the exit; and (d) the minimum path obtained from these two transforms.

The optimal path is determined by the sum of the distances (`D1 + D2`). The pixels belonging to the minimal trajectory are those for which this sum assumes its smallest value, defining a connection between entrance and exit with minimum geodesic length.

This principle allows solving mazes without the need to explicitly explore all possible routes. The solution emerges directly from the propagation of distances in a restricted domain. This approach is particularly relevant in highly complex mazes, such as those constructed from quasicrystalline structures and Hamiltonian cycles described by Singh (2024). In Zampirolli (2025), this same formalism is employed to solve a complex maze; below, the method is illustrated in a simplified version of the problem.

```
%%writefile tmp/fig_04_gdist_menor_caminho.cpp
#define MM_OUT "tmp/fig_04_gdist_menor_caminho.png"
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>

int main() {
    //| label: fig-04-gdist-menor-caminho
    //| fig-cap: "Menor caminho geodésico em um labirinto. As distâncias geodésicas são
    calculadas a partir da entrada e da saída usando mm.gdist. Os pixels cujo somatório das
    duas distâncias é igual à distância mínima entre os marcadores pertencem a um caminho
    ótimo."
    //| echo: true
    //| output: true

    // 1 = corredor, 0 = parede
```

```
mm::Image g(10, 10);
int g_data[10][10] = {
    {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}
};
for (int y = 0; y < 10; y++) {
    for (int x = 0; x < 10; x++) {
        g.at(y, x) = g_data[y][x];
    }
}

// marcador da entrada
mm::Image entrada(10, 10);
entrada.at(0, 1) = 1;

// marcador da saída
mm::Image saida(10, 10);
saida.at(9, 8) = 1;

// distâncias geodésicas
mm::Image D1 = mm::gdist(g, entrada);
mm::Image D2 = mm::gdist(g, saida);

// soma das distâncias
mm::Image S = mm::addm(D1, D2);

// menor valor válido da soma
int dmin = 255;
for (int i = 0; i < S.h * S.w; i++) {
    if (S.data[i] > 0 && S.data[i] < dmin) {
        dmin = S.data[i];
    }
}

// pixels pertencentes a um caminho ótimo
mm::Image caminho(10, 10);
for (int y = 0; y < 10; y++) {
    for (int x = 0; x < 10; x++) {
        caminho.at(y, x) = (S.at(y, x) == dmin) ? 255 : 0;
    }
}

std::cout << "Distância geodésica mínima: " << dmin << std::endl;

std::vector<std::string> titles = {
    "Labirinto",
    "Distância da Entrada",
    "Distância da Saída",
    "Menor Caminho\n(d=" + std::to_string(dmin) + ")"
};
mm::show(
    std::vector<mm::Image>{g, D1, D2, caminho},
    MM_OUT,
```

```
        titles,
        4
    );

    return 0;
}
```

```
Overwriting tmp/fig_04_gdist_menor_caminho.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_gdist_menor_caminho.cpp
-o tmp/fig_04_gdist_menor_caminho -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_gdist_menor_caminho \
  && test -f "tmp/fig_04_gdist_menor_caminho.png" \
  || echo "  mm::show não gravou tmp/fig_04_gdist_menor_caminho.png"
```

```
Distância geodésica mínima: 15
[1] Labirinto
[2] Distância da Entrada
[3] Distância da Saída
[4] Menor Caminho
(d=15)
```

```
try:
    mm.show(mm.read("tmp/fig_04_gdist_menor_caminho.png"), figsize=(14, 4))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_gdist_menor_caminho.png (ver a versao Python)")
```

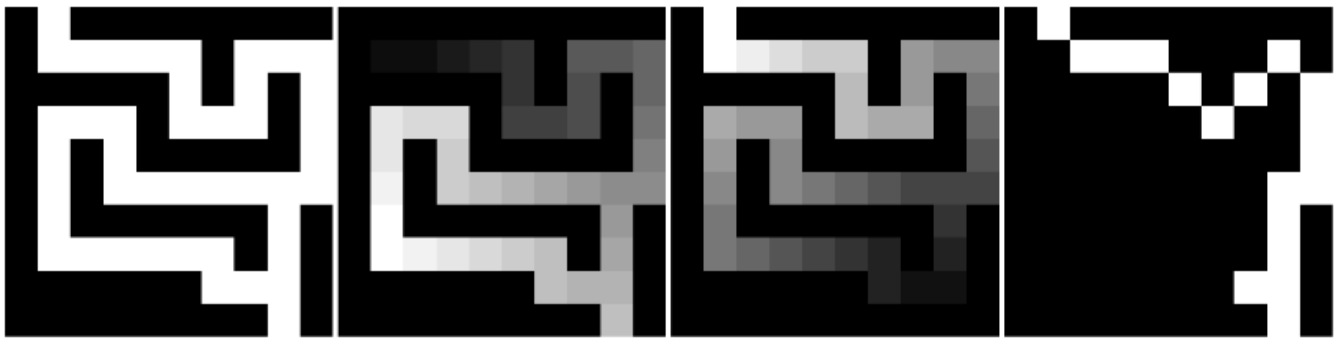


Figure 4.22: Menor caminho geodésico em um labirinto. As distâncias geodésicas são calculadas a partir da entrada e da saída usando mm.gdist. Os pixels cujo somatório das duas distâncias é igual à distância mínima entre os marcadores pertencem a um caminho ótimo.

4.4.4 Segmentation by *Watershed*

The ***Watershed*** algorithm interprets a grayscale image as a topographic surface, where high values correspond to mountains and low values correspond to valleys or catchment basins. In the context of marker-based segmentation, the maxima of the Distance Transform are often used to identify internal regions of objects, providing reliable seeds for the flooding process.

Segmentation is then performed through a conceptual simulation of progressive flooding from these markers. As the basins associated with different seeds expand, neighboring regions eventually come into contact. At that point, virtual barriers, known as *watershed lines*, are constructed, which then delimit the objects in the scene. This mechanism allows for the separation of adjacent or partially overlapping objects, even when they form a single connected component after thresholding.

The didactic implementation presented in this chapter initially explores the concept of region growing confined by a binary mask, as detailed in the interactive algorithm of Figure 4.23.

Didactic version versus classical implementation

The `mm::watershed0` function does not implement the classical *watershed* algorithm. Its purpose is to illustrate, in a simplified manner, the propagation of markers through region growing, allowing one to visualize how different seeds compete for the occupation of available space. The growth is delimited by a binary support mask and monitored by a stagnation control, producing a result similar to a Voronoi partition restricted to the geometry of the input objects.
The `mm::watershed` function, in turn, uses the optimized OpenCV implementation (`mm::watershed`), which performs flooding over a topographic surface defined by the input image. In this case, the propagation of markers is influenced by pixel values, causing separation lines to form naturally over the ridges of the relief.

Figure 4.24 presents an iterative simulator that illustrates the propagation of markers across the region of interest. Each marker acts as a flooding source that expands its area of influence until it encounters regions originating from other seeds. In the OpenCV algorithm, pixels belonging to the dividing lines are identified by the value `-1`, representing the boundaries between adjacent watersheds.

Morphological *Watershed* Pipeline

Marker-based *watershed* typically integrates into a broader segmentation workflow. In real images, preprocessing steps are often required to enhance contrast, reduce noise, and generate reliable markers. This complete workflow is summarized in Table 4.5.

Table 4.5: Complete marker-based *watershed* pipeline for real images.

Step	Operation	Purpose
1	CLAHE + Smoothing	Contrast enhancement and noise reduction
2	Thresholding	Initial separation between object and background
3	Opening/Closing	Removal of noise and small imperfections
4	Mask Dilation	Identification of Sure Background
5	Distance Transform + Threshold	Identification of Sure Foreground
6	Uncertain Region	Difference between Sure Background and Sure Foreground
7	`mm::watershed`	Propagation of markers through the uncertain region

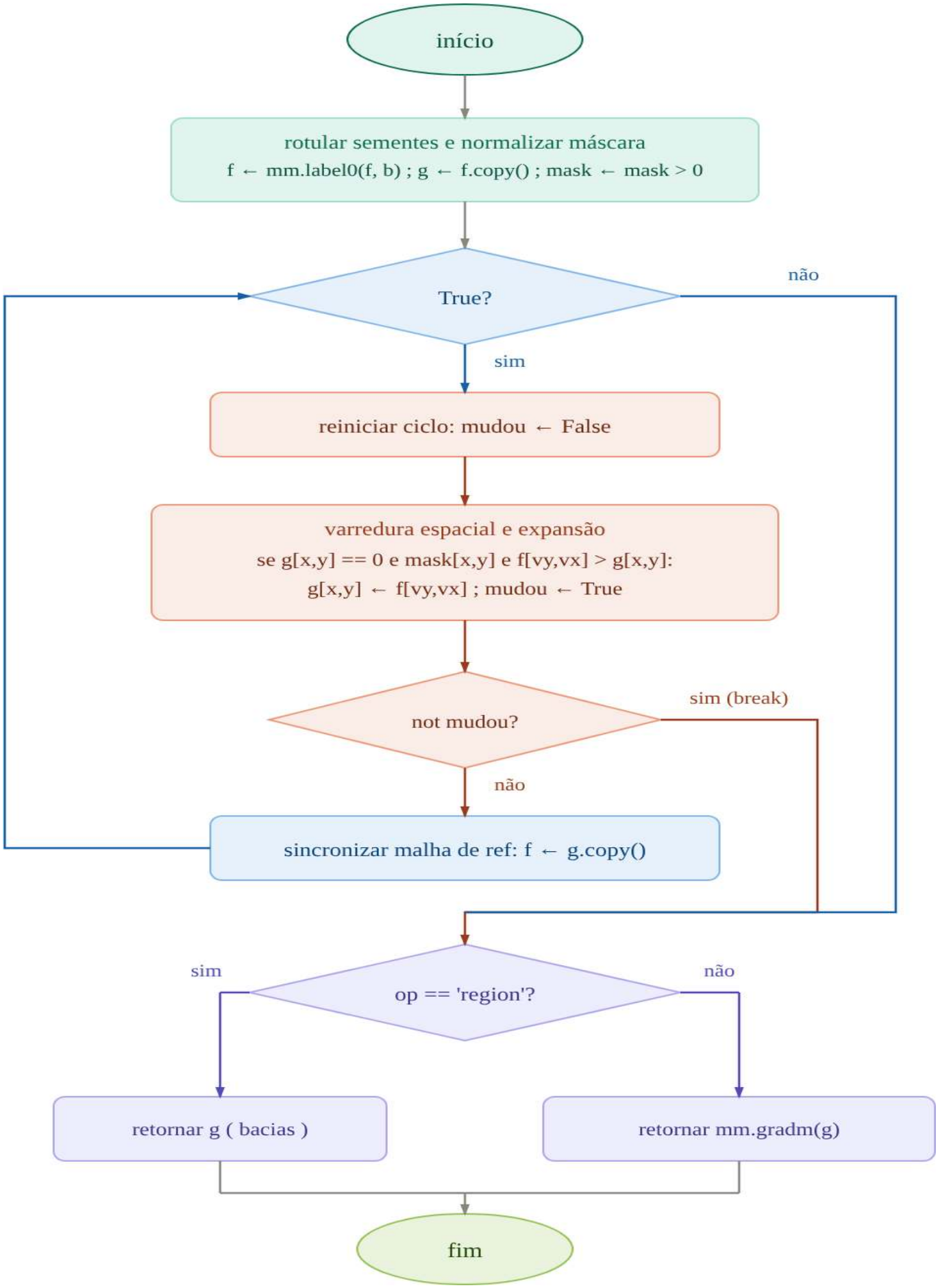
To focus exclusively on the concepts of Distance Transform, markers, and topographical flooding, the example in Figure 4.25 uses a synthetic binary image and adopts a simplified workflow, summarized in Table 4.6.

Table 4.6: Simplified pipeline used in the didactic example of Figure 4.25.

Step	Operation	Purpose
1	Distance Transform	Construction of the topographical surface
2	DT Threshold	Extraction of markers (Sure Foreground)

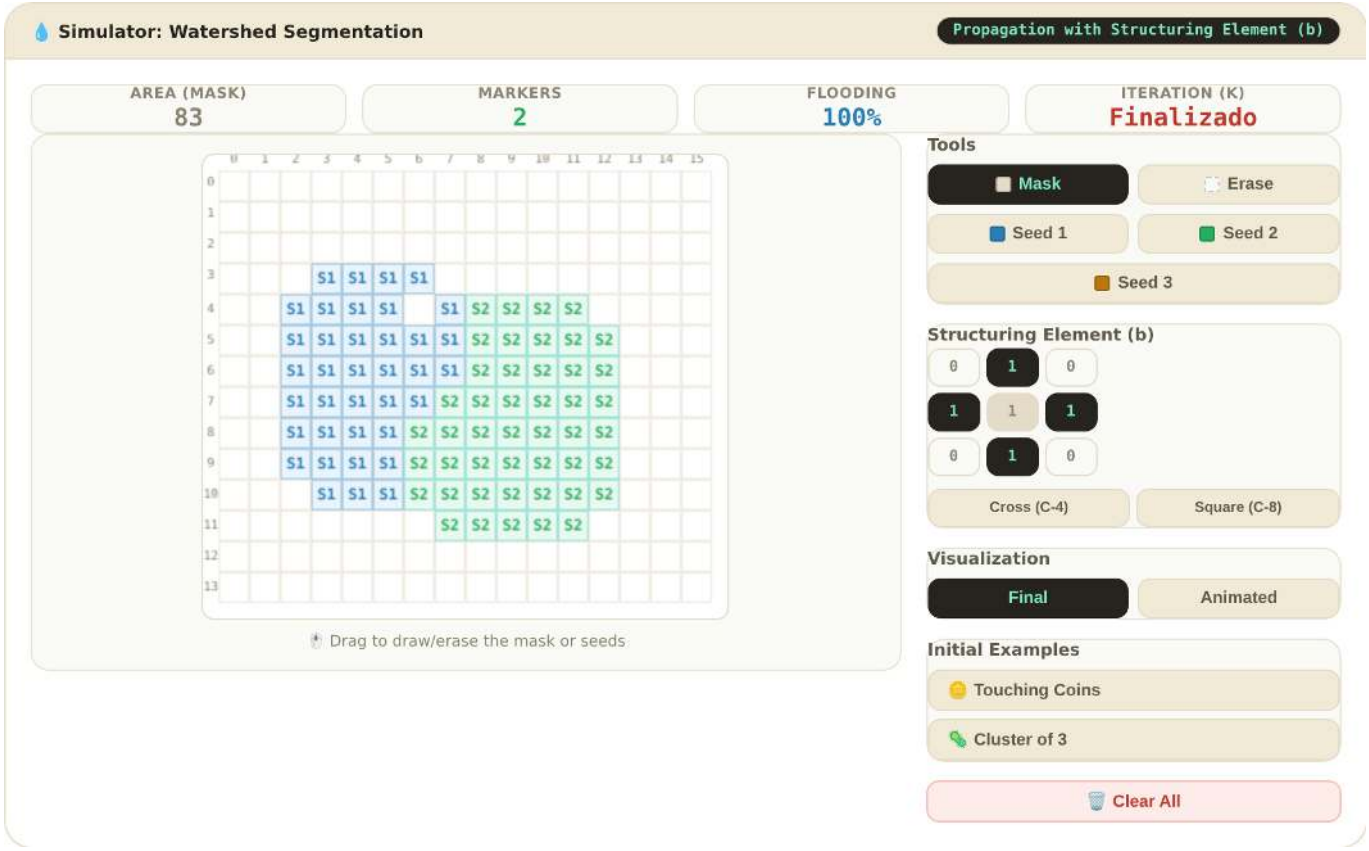
Step	Operation	Purpose
3	Dilation	Determination of Sure Background
4	Uncertain Region	Difference between background and markers
5	`mm::watershed`	Propagation of markers and generation of boundaries

```
%%writefile tmp/fig_04_watershed_didatico.cpp
#define MM_OUT "tmp/fig_04_watershed_didatico.png"
//| label: fig-04-watershed-didatico
//| fig-cap: "*Pipeline* *watershed* delimitado por máscara em imagem binária 20×20."
//| echo: true
//| output: true

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <filesystem>

int main() {
    // 1. Synthetic image and Distance Transform
    mm::Image f_sint(20, 20);
    f_sint = mm::circle(f_sint, 6, 10, 5, 255, -1);
    f_sint = mm::circle(f_sint, 14, 10, 5, 255, -1);
    mm::Image dist = mm::dist(f_sint);

    // 2. Markers (distance peaks)
    mm::Image m(20, 20);
    double maxDist = 0;
    for (int y = 0; y < dist.h; y++)
        for (int x = 0; x < dist.w; x++)
            maxDist = std::max(maxDist, (double)dist.at(y, x));
    for (int y = 0; y < dist.h; y++)
        for (int x = 0; x < dist.w; x++)
            m.at(y, x) = (dist.at(y, x) > 0.8 * maxDist) ? 255 : 0;

    // 3. Conditional Watershed0 execution (m=markers first, mask=f_sint)
    mm::Image w_reg  = mm::watershed0(m, f_sint, "region");
    mm::Image w_line = mm::watershed0(m, f_sint, "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, f_sint, "region");
    w_line = mm::watershed(m, f_sint, "line");

    // 4. Displaying the results
    mm::show({f_sint, dist, m, w_reg, w_line}, MM_OUT, {"Original", "Distância", "Marcadores",
"Regiões", "Linhas"}, 5);

// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(f_sint, "tmp/fig_04_watershed_didatico_0.png");
mm::write(dist, "tmp/fig_04_watershed_didatico_1.png");
mm::write(m, "tmp/fig_04_watershed_didatico_2.png");
mm::write(w_reg, "tmp/fig_04_watershed_didatico_3.png");
mm::write(w_line, "tmp/fig_04_watershed_didatico_4.png");
// [pdi:panel-io:end]
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-alg-watershed2.html

Figure 4.23: Didactic *Watershed* Algorithm by Region Growing Limited by Mask.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-04-watershed.html

Figure 4.24: Interactive simulator of the *Watershed* algorithm by morphological propagation. Draw the mask, position the markers and adjust the Structuring Element to observe the flooding. When basins meet simultaneously, the tie is resolved by randomly assuming one of the regions.

```
return 0;
}
```

```
Overwriting tmp/fig_04_watershed_didatico.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_watershed_didatico.cpp
-o tmp/fig_04_watershed_didatico -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_watershed_didatico \
  && test -f "tmp/fig_04_watershed_didatico.png" \
  || echo "  mm::show não gravou tmp/fig_04_watershed_didatico.png"
```

```
[1] Original
[2] Distância
[3] Marcadores
[4] Regiões
[5] Linhas
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_04_watershed_didatico_0.png"),
            mm.read("tmp/fig_04_watershed_didatico_1.png"),
            mm.read("tmp/fig_04_watershed_didatico_2.png"),
            mm.read("tmp/fig_04_watershed_didatico_3.png"),
            mm.read("tmp/fig_04_watershed_didatico_4.png"),
        ],
        titles=[
            'Original',
            'Distância',
            'Marcadores',
            'Regiões',
            'Linhas',
        ],
        cols=5,
        figsize=(16, 4),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_watershed_didatico_0.png (ver a versao Python)")
```

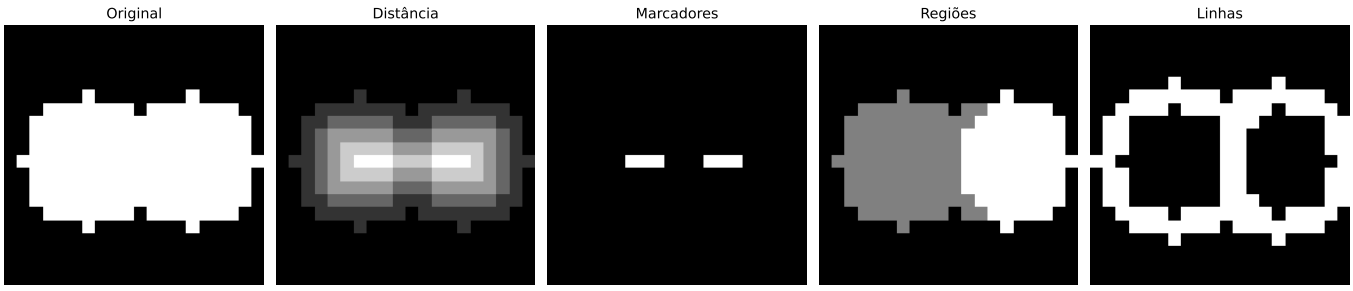


Figure 4.25: *Pipeline watershed* delimitado por máscara em imagem binária 20×20.

Application to overlapping coins:

```
%%writefile tmp/fig_04_watershed_moedas.cpp
#define MM_OUT "tmp/fig_04_watershed_moedas.png"
// Compile: g++ -std=c++17 -o program program.cpp -lopencv_core -lopencv_imgproc
-lopencv_highgui -lopencv_imgcodecs
```

```
#include "morph.hpp"
#include <opencv2/opencv.hpp>
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <set>
#include <cmath>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_coins_gray = mm::_read_state("tmp/state/img_coins_gray_8.png");
mm::Image img_final = mm::_read_state("tmp/state/img_final_55.png");
// [pdi:state-io:end]

    //| label: fig-04-watershed-moedas
    //| fig-cap: "*Pipeline* *Watershed* para separação de moedas sobrepostas: da máscara
    binária dilatada até os contornos finais anotados sobre a imagem original."
    //| echo: true
    //| output: true

    mm::Image img_base = img_coins_gray;

    // 1. Simulate overlapping/connected coins (using the base binary mask)
    mm::Image img_sobrepostas = mm::dil(img_final, mm::sebox(40));

    // 2. Morphological opening to clean noise
    mm::Image opening = mm::open(img_sobrepostas, mm::sebox(2));

    // 3. Distance Transform
    mm::Image dist = mm::dist(opening);
    mm::Image dist_vis(dist.h, dist.w);
    double dist_max = 0;
    for (int i = 0; i < dist.h * dist.w; i++) {
        if (dist.data[i] > dist_max) dist_max = dist.data[i];
    }
    for (int i = 0; i < dist.h * dist.w; i++) {
        if (dist_max > 0) {
            dist_vis.data[i] = static_cast<unsigned char>(255.0 * dist.data[i] / dist_max);
        } else {
            dist_vis.data[i] = dist.data[i];
        }
    }

    // 4. Safe peaks (Coin markers)
    mm::Image picos(dist.h, dist.w);
    for (int i = 0; i < dist.h * dist.w; i++) {
        if (dist_max > 0 && dist.data[i] > 0.5 * dist_max) {
            picos.data[i] = 255;
        } else {
            picos.data[i] = 0;
        }
    }

    // 5. Execute Watershed (Adjusted to use the new signature)
    // Pass 'opening' directly as mask, since it delimits the scope of expansion of the coins
    mm::Image ws_region = mm::watershed(picos, opening, "region");
    mm::Image ws_line = mm::watershed(picos, opening, "line");

    // 6. Object counting (ignores background 0)
```

```
    std::set<int> unique_labels;
    for (int i = 0; i < ws_region.h * ws_region.w; i++) {
        if (ws_region.data[i] > 0) unique_labels.insert(ws_region.data[i]);
    }
    std::vector<int> labels(unique_labels.begin(), unique_labels.end());
    std::cout << "Objetos detectados: " << labels.size() << "\n";

    // 7. Final Annotation
    cv::Mat img_base_mat(img_base.h, img_base.w, CV_8UC1, img_base.data.data());
    cv::Mat img_annotated;
    cv::cvtColor(img_base_mat, img_annotated, cv::COLOR_GRAY2BGR);

    for (size_t idx = 0; idx < labels.size(); idx++) {
        // Create mask for current region
        mm::Image mask_reg(ws_region.h, ws_region.w);
        long region_sum = 0;
        for (int i = 0; i < ws_region.h * ws_region.w; i++) {
            if (ws_region.data[i] == labels[idx]) {
                mask_reg.data[i] = 1;
                region_sum += 1;
            }
        }
        if (region_sum < 500) continue;

        // Calculate centroid
        double cy_sum = 0, cx_sum = 0;
        int count = 0;
        for (int y = 0; y < mask_reg.h; y++) {
            for (int x = 0; x < mask_reg.w; x++) {
                if (mask_reg.at(y, x) > 0) {
                    cy_sum += y;
                    cx_sum += x;
                    count++;
                }
            }
        }
        int cy = static_cast<int>(cy_sum / count);
        int cx = static_cast<int>(cx_sum / count);

        cv::putText(img_annotated, std::to_string(idx + 1), cv::Point(cx - 25, cy + 20),
                    cv::FONT_HERSHEY_SIMPLEX, 3.2, cv::Scalar(0, 255, 0), 8, cv::LINE_AA);
    }

    // Draw separation lines in red
    mm::Image kernel_ann(11, 11);
    std::fill(kernel_ann.data.begin(), kernel_ann.data.end(), 1);
    mm::Image mask_ann = mm::dil(ws_line, kernel_ann);

    // Apply red color where mask > 0
    for (int y = 0; y < mask_ann.h; y++) {
        for (int x = 0; x < mask_ann.w; x++) {
            if (mask_ann.at(y, x) > 0) {
                img_annotated.at<cv::Vec3b>(y, x) = cv::Vec3b(255, 0, 0);
            }
        }
    }

    // Convert back to mm::Image for display
    mm::Image img_annotated_mm(img_annotated.rows, img_annotated.cols, 3);
    std::memcpy(img_annotated_mm.data.data(), img_annotated.data,
img_annotated_mm.data.size());
```

```
    // Copy img_sobrepostas to mm::Image for ws_region display (single channel)
    mm::Image ws_region_mm = ws_region;
    mm::Image picos_mm = picos;

    // Display
    mm::show(
        std::vector<mm::Image>{img_base, img_sobrepostas, dist_vis, picos_mm, ws_region_mm,
img_annotated_mm},
        MM_OUT,
        std::vector<std::string>{"Original", "Sobrepostas", "Distância", "Marcadores",
"Watershed", "Anotado"},
        3
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_base, "tmp/fig_04_watershed_moedas_0.png");
mm::write(img_sobrepostas, "tmp/fig_04_watershed_moedas_1.png");
mm::write(dist_vis, "tmp/fig_04_watershed_moedas_2.png");
mm::write(picos, "tmp/fig_04_watershed_moedas_3.png");
mm::write(ws_region, "tmp/fig_04_watershed_moedas_4.png");
mm::write(img_annotated, "tmp/fig_04_watershed_moedas_5.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_04_watershed_moedas.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_watershed_moedas.cpp -o
tmp/fig_04_watershed_moedas -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_watershed_moedas \
  && test -f "tmp/fig_04_watershed_moedas.png" \
  || echo "  mm::show não gravou tmp/fig_04_watershed_moedas.png"
```

```
Objetos detectados: 12
[1] Original
[2] Sobrepostas
[3] Distância
[4] Marcadores
[5] Watershed
[6] Anotado
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_04_watershed_moedas_0.png"),
            mm.read("tmp/fig_04_watershed_moedas_1.png"),
            mm.read("tmp/fig_04_watershed_moedas_2.png"),
            mm.read("tmp/fig_04_watershed_moedas_3.png"),
            mm.read("tmp/fig_04_watershed_moedas_4.png"),
            mm.read("tmp/fig_04_watershed_moedas_5.png"),
        ],
        titles=[
            'Original',
            'Sobrepostas',
            'Distância',
            'Marcadores',
            'Watershed',
```

```
            'Anotado',
        ],
        cols=3,
        rows=2,
        figsize=(12, 8),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_watershed_moedas_0.png (ver a versao Python)")
```

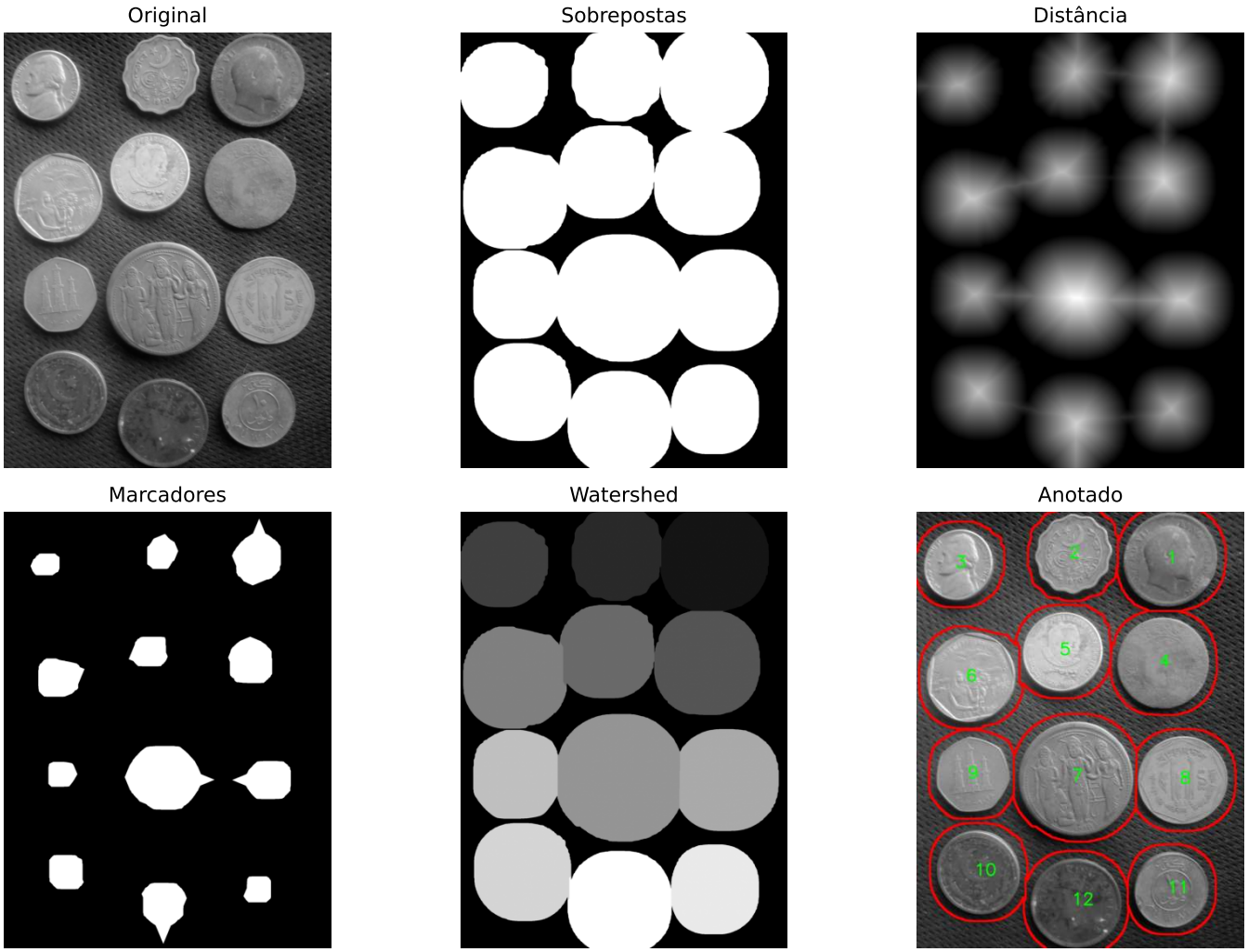


Figure 4.26: *Pipeline Watershed* para separação de moedas sobrepostas: da máscara binária dilatada até os contornos finais anotados sobre a imagem original.

4.5 Shape Component Extraction and Descriptors

After segmentation and morphological refinement, the next step consists of identifying each object present in the image individually and extracting its geometric properties. This stage is fundamental for measurement, classification, and pattern recognition tasks.

A **connected component** is a maximal set of pixels belonging to the object that remain mutually connected according to a previously defined connectivity relation (4- or 8-connectivity). After labeling, each component receives a unique identifier, allowing its characteristics to be analyzed individually.

OpenCV offers two complementary approaches for this analysis, summarized in Table 4.7.

Table 4.7: Comparison between approaches based on connected components and contours.

	`connectedComponentsWithStats`	`findContours`
Returns	label per pixel and statistics per component	sequence of points describing the border
Direct descriptors	area, bounding box, and centroid	perimeter, shape, and hierarchy
Objects in contact	tends to merge connected regions	tends to produce a single external contour
Typical use	counting, filtering, and labeling	geometric analysis and shape descriptors

4.5.1 Labeling and Component Statistics

`mm::label0` assigns a label to each connected component. From the labeled image, the **area** (pixel count) and the **bounding box** of each object are obtained by scanning (Figure 4.27). OpenCV's `connectedComponentsWithStats`, which returns these statistics ready-made, and the colored annotations on the image are handled in the Python track.

```
%%writefile tmp/fig_04_componentes.cpp
#define MM_OUT "tmp/fig_04_componentes.png"
// Compile with: g++ -std=c++17 -DMM_USE_OPENCV -o program program.cpp $(pkg-config --cflags
--libs opencv4) -lm

#include "morph.hpp"
#include <opencv2/opencv.hpp>
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_coins_gray = mm::_read_state("tmp/state/img_coins_gray_8.png");
mm::Image img_final = mm::_read_state("tmp/state/img_final_55.png");
// [pdi:state-io:end]

    // 1. Labeling and statistics
    cv::Mat labels, stats, centroids;
    cv::Mat img_final_mat(img_final.h, img_final.w, CV_8UC1, img_final.data.data());
    int n = cv::connectedComponentsWithStats(img_final_mat, labels, stats, centroids, 8);

    // 2. Component coloring - FIXED palette (generated once with
    // np.random.seed(4)) so the C++ track can reproduce it color by color
    // without relying on numpy's generator. If n exceeds len(PALETA), cycles.
    std::vector<std::vector<int>> PALETA = {
        {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}};

    mm::Image img_colored(labels.rows, labels.cols, 3);
    mm::Image img_annotated(labels.rows, labels.cols, 3);

    for (int y = 0; y < labels.rows; y++) {
        for (int x = 0; x < labels.cols; x++) {
            int label = labels.at<int>(y, x);
            int color_idx = label % PALETA.size();
```

```
            for (int c = 0; c < 3; c++) {
                img_colored.at(y, x, c) = PALETA[color_idx][c];
                img_annotated.at(y, x, c) = PALETA[color_idx][c];
            }
        }
    }

    // 3. Descriptor table
    std::cout << "Componentes detectados (excluindo fundo): " << (n - 1) << "\n";
    std::cout << std::setw(4) << "ID" << std::setw(8) << "Área" << std::setw(6) << "cx"
              << std::setw(6) << "cy" << std::setw(6) << "w" << std::setw(6) << "h" << "\n";
    std::cout << std::string(42, '-') << "\n";

    for (int i = 1; i < n; i++) {
        int area = stats.at<int>(i, cv::CC_STAT_AREA);
        int cx = static_cast<int>(centroids.at<double>(i, 0));
        int cy = static_cast<int>(centroids.at<double>(i, 1));
        int w = stats.at<int>(i, cv::CC_STAT_WIDTH);
        int h = stats.at<int>(i, cv::CC_STAT_HEIGHT);

        std::cout << std::setw(4) << i << std::setw(8) << area << std::setw(6) << cx
                  << std::setw(6) << cy << std::setw(6) << w << std::setw(6) << h << "\n";

        // Convert annotated image to cv::Mat for text drawing
        cv::Mat img_ann_mat(img_annotated.h, img_annotated.w, CV_8UC3,
img_annotated.data.data());

        for (const auto& [cor, esp] : std::vector<std::pair<cv::Scalar,
        int>>{{cv::Scalar(0,0,0), 10}, {cv::Scalar(0,0,255), 5}}) {
            cv::putText(img_ann_mat, std::to_string(i) + ": " + std::to_string(area),
                        cv::Point(cx - 150, cy + 18),
                        cv::FONT_HERSHEY_SIMPLEX, 2.0, cor, esp, cv::LINE_AA);
        }
    }

    mm::show(
        std::vector<mm::Image>{img_coins_gray, img_final, img_colored, img_annotated},
        MM_OUT,
        std::vector<std::string>{"Original", "Segmentação Final", "Componentes Conexos",
"Áreas Anotadas"},
        4
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_coins_gray, "tmp/fig_04_componentes_0.png");
mm::write(img_final, "tmp/fig_04_componentes_1.png");
mm::write(img_colored, "tmp/fig_04_componentes_2.png");
mm::write(img_annotated, "tmp/fig_04_componentes_3.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_04_componentes.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_componentes.cpp -o
tmp/fig_04_componentes -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_componentes \
  && test -f "tmp/fig_04_componentes.png" \
```

```
  || echo "  mm::show não gravou tmp/fig_04_componentes.png"
```

```
Componentes detectados (excluindo fundo): 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
[1] Original
[2] Segmentação Final
[3] Componentes Conexos
[4] Áreas Anotadas
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_04_componentes_0.png"),
            mm.read("tmp/fig_04_componentes_1.png"),
            mm.read("tmp/fig_04_componentes_2.png"),
            mm.read("tmp/fig_04_componentes_3.png"),
        ],
        titles=[
            'Original',
            'Segmentação Final',
            'Componentes Conexos',
            'Áreas Anotadas',
        ],
        cols=4,
        figsize=(18, 6),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_04_componentes_0.png (ver a versao Python)")
```

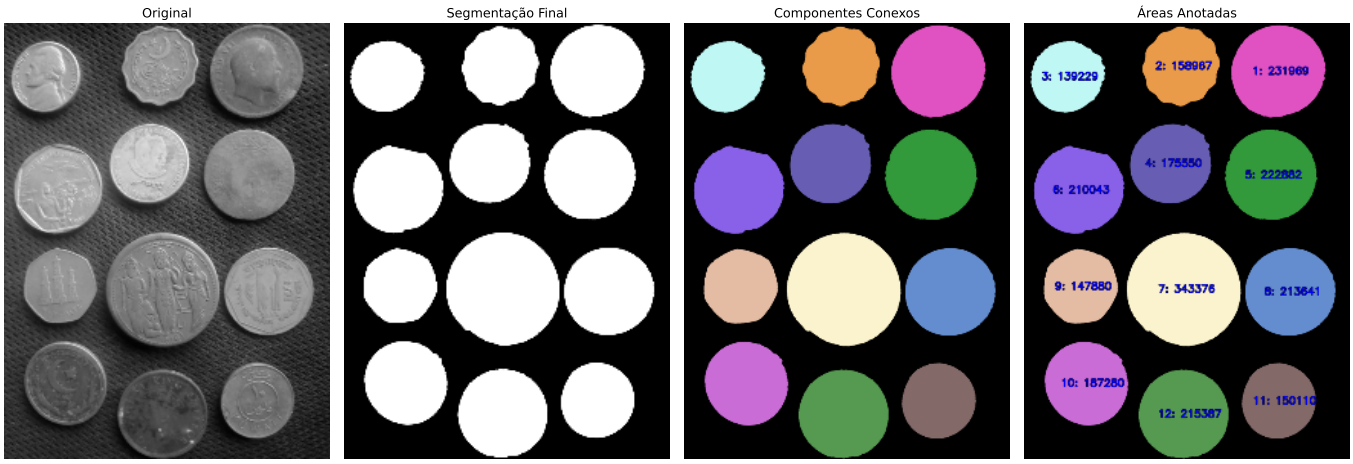


Figure 4.27: Componentes conexos extraídos após o *pipeline* CLAHE → Otsu → limpeza morfológica. Cada objeto é colorido com cor distinta e anotado com sua área em pixels.

4.5.2 Shape Descriptors

Descriptors based on **vector contour** — perimeter (`arcLength`), polygon area (`contourArea`), circularity, moments, and polygonal approximation (`approxPolyDP`) — depend on boundary tracing (`findContours`), which is absent in `morph.hpp`. They remain only in the Python track. In the C++ track, the **morphological gradient** (`mm::gradm`) highlights the contour of each object (Figure 4.28).

```
%%writefile tmp/fig_04_contornos.cpp
#define MM_OUT "tmp/fig_04_contornos.png"
// Compile: g++ -std=c++17 -o program program.cpp -lopencv_core -lopencv_imgproc
-lopencv_imgcodecs $(pkg-config --cflags --libs opencv4)
#include "morph.hpp"
#include <opencv2/opencv.hpp>
#include <iostream>
#include <vector>
#include <cmath>
#include <iomanip>
#include <numeric>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_coins_gray = mm::_read_state("tmp/state/img_coins_gray_8.png");
mm::Image img_final = mm::_read_state("tmp/state/img_final_55.png");
// [pdi:state-io:end]

    //| label: fig-04-contornos
    //| fig-cap: "Contornos extraídos com *findContours*. Cada moeda é anotada com sua
    circularidade - valores próximos de 1 confirmam forma circular."
    //| echo: true
    //| output: true

    // Bridge mm::Image to cv::Mat
    cv::Mat img_final_mat(img_final.h, img_final.w, CV_8UC1, img_final.data.data());

    // Find contours
    std::vector<std::vector<cv::Point>> contornos;
    cv::findContours(img_final_mat, contornos, cv::RETR_EXTERNAL, cv::CHAIN_APPROX_SIMPLE);

    // Prepare output images
    cv::Mat gray_mat(img_coins_gray.h, img_coins_gray.w, CV_8UC1, img_coins_gray.data.data());
    cv::Mat img_contornos_mat;
```

```
    cv::cvtColor(gray_mat, img_contornos_mat, cv::COLOR_GRAY2BGR);
    cv::Mat img_circulares_mat = img_contornos_mat.clone();

    std::cout << "Contornos detectados: " << contornos.size() << "\n";
    std::cout << std::setw(4) << "ID" << std::setw(8) << "Área" << std::setw(10) <<
"Perímetro"
              << std::setw(14) << "Circularidade" << "\n";
    std::cout << std::string(42, '-') << "\n";

    int id = 1;
    for (const auto& cnt : contornos) {
        double area = cv::contourArea(cnt);
        double perim = cv::arcLength(cnt, true);
        double circ = (perim > 0) ? (4 * M_PI * area / (perim * perim)) : 0;
        cv::Moments M = cv::moments(cnt);
        int cx = (M.m00 > 0) ? static_cast<int>(M.m10 / M.m00) : 0;
        int cy = (M.m00 > 0) ? static_cast<int>(M.m01 / M.m00) : 0;

        std::cout << std::setw(4) << id << std::setw(8) << std::fixed << std::setprecision(0)
                  << area << std::setw(10) << std::setprecision(1) << perim
                  << std::setw(14) << std::setprecision(3) << circ << "\n";

        cv::drawContours(img_contornos_mat, std::vector<std::vector<cv::Point>>{cnt}, -1,
                         cv::Scalar(0, 255, 0), 3);
        cv::drawContours(img_circulares_mat, std::vector<std::vector<cv::Point>>{cnt}, -1,
                         cv::Scalar(0, 255, 0), 3);

        // Draw text with two layers for outline effect
        std::vector<std::pair<cv::Scalar, int>> styles = {
            {cv::Scalar(0, 0, 0), 8},
            {cv::Scalar(255, 0, 0), 3}
        };
        for (const auto& [color, thickness] : styles) {
            cv::putText(img_circulares_mat, std::to_string(static_cast<int>(circ * 100) /
100.0),
                        cv::Point(cx - 80, cy + 15),
                        cv::FONT_HERSHEY_SIMPLEX, 3.6, color, thickness, cv::LINE_AA);
        }
        id++;
    }

    // Convert back to mm::Image
    mm::Image img_contornos_out(img_contornos_mat.rows, img_contornos_mat.cols,
img_contornos_mat.channels());
    std::memcpy(img_contornos_out.data.data(), img_contornos_mat.data,
img_contornos_out.data.size());

    mm::Image img_circulares_out(img_circulares_mat.rows, img_circulares_mat.cols,
img_circulares_mat.channels());
    std::memcpy(img_circulares_out.data.data(), img_circulares_mat.data,
img_circulares_out.data.size());

    mm::show(
        std::vector<mm::Image>{img_final, img_contornos_out, img_circulares_out},
        MM_OUT,
        std::vector<std::string>{"Segmentação Final", "Contornos", "Circularidade"},
        3
    );

    return 0;
}
```

```
Overwriting tmp/fig_04_contornos.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_contornos.cpp -o
tmp/fig_04_contornos -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_contornos \
  && test -f "tmp/fig_04_contornos.png" \
  || echo "  mm::show não gravou tmp/fig_04_contornos.png"
```

```
Contornos detectados: 12
  ID   ÁreaPerímetro 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
[1] Segmentação Final
[2] Contornos
[3] Circularidade
```

```
try:
    mm.show(mm.read("tmp/fig_04_contornos.png"), figsize=(18, 6))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_04_contornos.png
    (ver a versao Python)")
```

Figure 4.28: Contornos extraídos com *findContours*. Cada moeda é anotada com sua circularidade — valores próximos de 1 confirmam forma circular.

4.5.3 Connection with Modern Object Detection

The descriptors extracted in the previous sections — especially *bounding boxes*, centroids, areas, and shape measures — establish a natural bridge between classical morphological segmentation and modern object detection systems. Although the techniques studied in this chapter use operations on pixels and segmented regions, many of the representations produced are directly compatible with the formats employed in contemporary computer vision models.

Deep learning-based detectors, such as the YOLO (*You Only Look Once*) family (REDMON, 2016), operate directly on color images and produce, for each detected object, a *bounding box* described by the center (cx, cy) and the dimensions (w, h), as well as a class and a confidence score. This representation shares the same basic geometric structure obtained by `connectedComponentsWithStats`, although it is produced by a learned model rather than by explicit segmentation.

Figure 4.29 illustrates how *bounding boxes* obtained by morphology can be exported in the YOLO format to compose datasets used in training or evaluating detectors.

```
%%writefile tmp/fig_04_bbox.cpp
#define MM_OUT "tmp/fig_04_bbox.png"
//| label: fig-04-bbox
//| fig-cap: "*Bounding boxes* derivadas dos componentes conexos sobrepostas à imagem
original. As anotações são exportadas no formato YOLO (*classe cx cy w h*), com coordenadas
normalizadas para o intervalo [0,1]."
//| echo: true
//| output: true
#include "morph.hpp"
#include <opencv2/opencv.hpp>
#include <iostream>
#include <vector>
#include <string>
#include <cstdio>
#include <cstring>
#include <fstream>
#include <iomanip>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_coins_gray = mm::_read_state("tmp/state/img_coins_gray_8.png");
mm::Image img_final = mm::_read_state("tmp/state/img_final_55.png");
// [pdi:state-io:end]

    // Recalcula rótulos/estatísticas a partir da segmentação final
    cv::Mat img_final_mat(img_final.h, img_final.w, CV_8UC1, img_final.data.data());
    cv::Mat img_coins_gray_mat(img_coins_gray.h, img_coins_gray.w, CV_8UC1,
img_coins_gray.data.data());

    cv::Mat labels, stats, centroids;
    int n = cv::connectedComponentsWithStats(img_final_mat, labels, stats, centroids, 8);

    int H_img = img_coins_gray.h;
    int W_img = img_coins_gray.w;
    cv::Mat img_bbox_cv;
    cv::cvtColor(img_coins_gray_mat, img_bbox_cv, cv::COLOR_GRAY2BGR);

    int CLASSE = 0;   // 0 = moeda (única categoria neste exemplo)

    std::cout << std::setw(4) << "cls" << std::setw(8) << "cx_n" << std::setw(8) << "cy_n" <<
std::setw(8) << "w_n" << std::setw(8) << "h_n" << "  ← formato YOLO\n";
    std::cout << std::string(54, '-') << "\n";

    std::vector<std::string> yolo_linhas;
    for (int i = 1; i < n; i++) {
        int x0 = stats.at<int>(i, cv::CC_STAT_LEFT);
        int y0 = stats.at<int>(i, cv::CC_STAT_TOP);
        int w  = stats.at<int>(i, cv::CC_STAT_WIDTH);
        int h  = stats.at<int>(i, cv::CC_STAT_HEIGHT);

        double cx_n = (x0 + w / 2.0) / W_img;
```

```
        double cy_n = (y0 + h / 2.0) / H_img;
        double w_n  = w / double(W_img);
        double h_n  = h / double(H_img);

        char linha[100];
        snprintf(linha, sizeof(linha), "%d %.4f %.4f %.4f %.4f", CLASSE, cx_n, cy_n, w_n,
h_n);
        yolo_linhas.push_back(linha);

        printf("%4d %8.4f %8.4f %8.4f %8.4f\n", CLASSE, cx_n, cy_n, w_n, h_n);

        cv::rectangle(img_bbox_cv, cv::Point(x0, y0), cv::Point(x0 + w, y0 + h), cv::Scalar(0,
255, 0), 4);
        cv::putText(img_bbox_cv, "moeda", cv::Point(x0 + 8, y0 + 60),
                    cv::FONT_HERSHEY_SIMPLEX, 3.8, cv::Scalar(255, 0, 0), 5, cv::LINE_AA);
    }

    // Exportar arquivo de anotação no formato YOLO
    std::ofstream f("moedas.txt");
    for (size_t i = 0; i < yolo_linhas.size(); i++) {
        if (i > 0) f << "\n";
        f << yolo_linhas[i];
    }
    f.close();
    std::cout << "\nAnotação salva em moedas.txt\n";

    // Convert cv::Mat back to mm::Image
    mm::Image img_bbox(img_bbox_cv.rows, img_bbox_cv.cols, img_bbox_cv.channels());
    std::memcpy(img_bbox.data.data(), img_bbox_cv.data, img_bbox.data.size());

    mm::show(std::vector<mm::Image>{img_coins_gray, img_final, img_bbox},
             MM_OUT,
             std::vector<std::string>{"Original", "Segmentação Final", "Bounding Boxes
(formato YOLO)"},
             3);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_coins_gray, "tmp/fig_04_bbox_0.png");
mm::write(img_final, "tmp/fig_04_bbox_1.png");
mm::write(img_bbox, "tmp/fig_04_bbox_2.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_04_bbox.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_04_bbox.cpp -o
tmp/fig_04_bbox -lopencv_imgproc -lopencv_imgcodecs -lopencv_core \
  && ./tmp/fig_04_bbox \
  && test -f "tmp/fig_04_bbox.png" \
  || echo "  mm::show não gravou tmp/fig_04_bbox.png"
```

```
 cls    cx_n    cy_n     w_n     h_n  ← formato YOLO
-----------------------------------------------------
   0   0.7753   0.1102   0.2880   0.2117
   0   0.4784   0.0984   0.2359   0.1828
   0   0.1331   0.1225   0.2255   0.1637
   0   0.4464   0.3217   0.2469   0.1816
```

```
   0   0.7523   0.3471   0.2807   0.2074
   0   0.1664   0.3812   0.2745   0.2016
   0   0.4862   0.6104   0.3474   0.2598
   0   0.8115   0.6154   0.2760   0.2012
   0   0.1724   0.6023   0.2250   0.1711
   0   0.1893   0.8258   0.2536   0.1922
   0   0.7763   0.8652   0.2255   0.1734
   0   0.4854   0.8953   0.2750   0.2039

Anotação salva em moedas.txt
[1] Original
[2] Segmentação Final
[3] Bounding Boxes (formato YOLO)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_04_bbox_0.png"),
            mm.read("tmp/fig_04_bbox_1.png"),
            mm.read("tmp/fig_04_bbox_2.png"),
        ],
        titles=[
            'Original',
            'Segmentação Final',
            'Bounding Boxes (formato YOLO)',
        ],
        cols=3,
        figsize=(18, 6),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_04_bbox_0.png (ver
    a versao Python)")
```

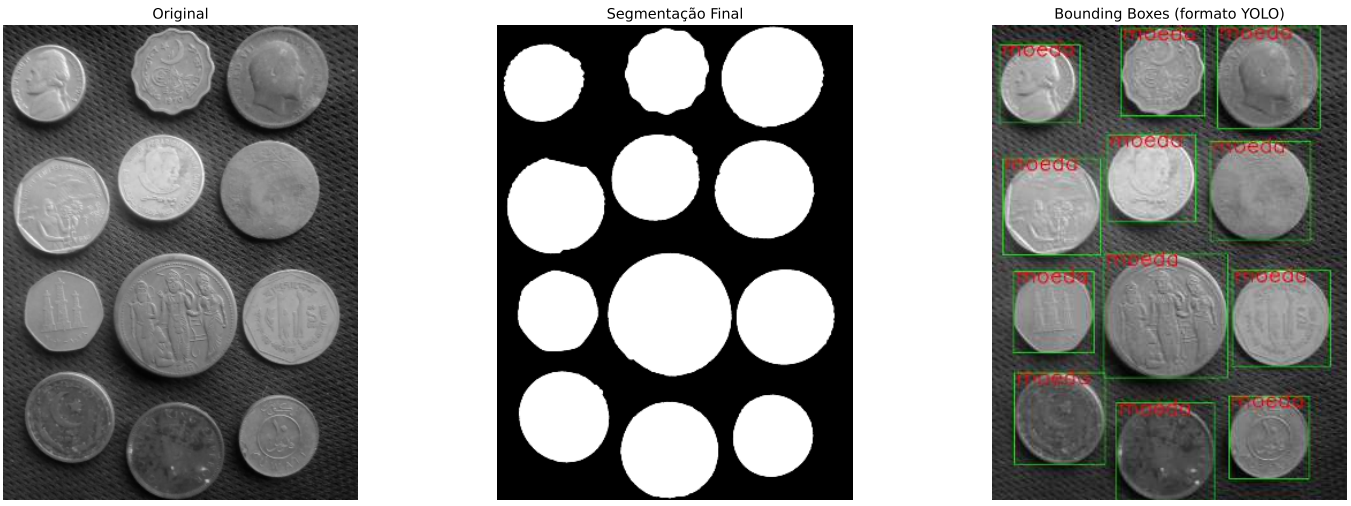


Figure 4.29: *Bounding boxes* derivadas dos componentes conexos sobrepostas à imagem original. As anotações são exportadas no formato YOLO (*classe cx cy w h*), com coordenadas normalizadas para o intervalo [0,1].

The YOLO format stores each object in a line containing five fields:

```
classe cx cy w h
```

where (cx, cy) represents the center of the *bounding box* and (w, h) its dimensions. All geometric values are normalized to the interval $[0, 1]$ with respect to the image's width and height. The **class** is an integer identifier

associated with a category defined by the dataset (e.g., `0 → coin`). When there are multiple categories — such as gold coin (`0`), silver coin (`1`), and plastic disk (`2`) — one simply assigns the corresponding identifier to each object before export, maintaining exactly the same annotation format. In the previous example, this information was stored in the `moedas.txt` file.

The workflow presented in this chapter — segmentation → labeling → *bounding box* extraction — conceptually corresponds to the **annotation** (*labeling*) step employed in building training sets for modern detectors. Specialized tools, such as Label Studio and Roboflow, automate this process in complex images, but the fundamental logic remains the same: associating each object with a region of interest and a class. In controlled scenarios, with a uniform background and well-separated objects, morphological techniques can even generate annotations automatically or serve as a starting point for manual labeling, significantly reducing the effort of dataset construction. In more complex real-world applications, however, human validation remains necessary to ensure the quality of annotations.

Evaluation: IoU (*Intersection over Union*)

A simple way to evaluate the quality of a segmentation is to compare it against a reference mask (*ground truth*). The most widely used metric for this purpose is **IoU** (*Intersection over Union*):

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} \tag{4.16}$$

where A represents the segmentation produced by the algorithm and B the reference segmentation. The IoU value ranges between 0 and 1. The higher the value, the greater the overlap between the masks. An IoU of 1 indicates perfect correspondence between the obtained segmentation and the reference. The same metric is also extensively used in object detection, applied to predicted and annotated *bounding boxes*. In this field, IoU values greater than 0.5 are often adopted as the minimum criterion to consider a detection correct.

Beyond Morphology

The techniques studied in this chapter segment objects by exploiting spatial connectivity, morphological operations, and topographic relief. However, there are alternative approaches based on feature clustering, such as the *k-means* algorithm, Gaussian mixture models (GMM), and more recent methods based on deep learning. These techniques will be revisited in Part II of the book, dedicated to Computer Vision.

4.6 Summary

This chapter presented the main segmentation and mathematical morphology techniques, concluding the study of DIP in the spatial domain.

- **Preprocessing and thresholding:** The combination of adaptive CLAHE equalization and Otsu's method proved effective in inducing bimodal separation in the histogram and simplifying the binarization of images with non-uniform illumination.
- **Erosion and dilation:** Fundamental morphological operators based on the search for local minima and maxima in a neighborhood defined by the structuring element B. These are dual operators by complement and were implemented using the functions `mm::ero` and `mm::dil`.
- **Opening and closing:** Compositions of erosion and dilation that allow removing noise, smoothing contours, and filling small gaps while preserving the overall structure of objects.
- **Morphological reconstruction:** An iterative geodesic process that propagates a marker within the limits imposed by a mask, forming the basis of operators such as `mm::clohole` and `mm::edgeoff`.
- **Binary cleaning pipeline:** A consolidated workflow composed of CLAHE → Otsu → opening → `mm::clohole` → restricted opening → `mm::edgeoff`, producing suitable masks for quantitative analysis.
- **Grayscale morphology:** An algebraic extension based on weighted minima and maxima, enabling operators such as morphological gradient and *top-hat* filters for enhancing local structures.

- **Distance Transform and Watershed:** The Distance Transform allowed generating automatic markers for the *watershed* algorithm, enabling the separation of adjacent or partially overlapping objects.
- **Connected components and descriptors:** Region labeling (`mm::label0`) and contour extraction (contour extraction) allowed computing geometric descriptors such as area, centroid, perimeter, circularity, and bounding boxes.
- **Connection with Modern Computer Vision:** The bounding boxes extracted by morphology were exported in YOLO format, highlighting the link between classical segmentation techniques and modern object detection systems.

Chapter 5 will introduce processing techniques in the **frequency domain**, addressing the **Fourier Transform**, spectral filtering, and the fundamentals of **image compression**, including DCT, JPEG, and wavelets.

4.7 🤖 Using Gemini Notebook as a Complementary Tutor

In this edition, the use of **Gemini Notebook** is encouraged as a complementary learning tool. Based on artificial intelligence, the system uses exclusively the documents provided by the author as its source of knowledge, producing responses aligned with the content and approach adopted throughout this chapter.

🎓 Study with the Intelligent Tutor

🚀 ACCESS GEMINI NOTEBOOK: CHAPTER 04

🌐 Language and Programming Language

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

⚠️ Notice Regarding AI-Generated Content

Although it is a valuable study support tool, Gemini Notebook may occasionally produce incomplete, inaccurate, or incorrect responses. It is recommended to validate the information by consulting the chapter material, books, scientific articles, and other reliable academic sources. Whenever possible, run and experiment with the practical examples presented throughout the text to consolidate your understanding of the concepts.

4.8 Exercise List

1. **(10%)** Manually implement Otsu's criterion without using `mm::threshold`. Compute the interclass variance $\sigma_B^2(T)$ for all thresholds $T \in [0, 255]$ using `mm::hist`, identify the optimal threshold T^* and compare the result with the value obtained by OpenCV. Plot σ_B^2 as a function of T and highlight the maximum point.

2. **(15%)** Apply adaptive thresholding with block sizes of 11, 31, and 51 to an image containing non-uniform illumination. Compare the results with global Otsu thresholding and discuss the advantages and limitations of each approach.

3. **(15%)** Run the watershed *pipeline* on the coins image, varying the threshold applied to the Distance Transform (0.3, 0.5, and 0.7 times the maximum value). Explain how this parameter influences marker generation, the separation of adjacent objects, and the occurrence of over-segmentation.

4. **(15%)** Using `mm::drawImg`, construct a step-by-step visual demonstration of the erosion of a 7×7 binary image with a 3×3 square structuring element. For each analyzed position, indicate whether the

structuring element is fully contained within the object and justify the value assigned to the output pixel.

5. **(15%)** Experimentally demonstrate the duality between erosion and dilation by verifying the identity $(A \ominus B)^c = A^c \oplus \hat{B}$ using `mm::ero`, `mm::dil`, and `mm::bnot`. Compute the pixel-by-pixel difference between the two sides of the equation and present the result using `mm::histImg` or equivalent visualization.

6. **(15%)** Manually implement the morphological gradient using only `mm::ero` and `mm::dil`, comparing the result with `mm::gradm(img, B)`. Evaluate the effect of different structuring elements (3×3 square, 5×5 disk, and 1×9 line) on edge detection.

7. **(15%)** Construct a complete *pipeline* for counting and classifying coins by size (small, medium, and large) using area and circularity as descriptors. Manually generate a reference mask (*ground truth*) and compute the IoU metric (*Intersection over Union*) to assess segmentation quality. Present the results in a table and through visualizations produced with `mm::show`.

Chapter References

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

- Gonzalez (2018) for the concepts of segmentation, Otsu thresholding, Distance Transform, *watershed*, mathematical morphology, and shape descriptors.
- Matheron (1975) and Serra (1982) for the original theoretical foundation, algebraic formulation, and development of Mathematical Morphology.
- Szeliski (2022) for region-based segmentation, connected component labeling, marker-controlled *watershed*, and segmentation evaluation using the IoU metric.
- Bradski (2008) for the practical use of the OpenCV library, including functions such as `mm::dist`, `mm::watershed`, `mm::label0`, and contour extraction.
- Redmon (2016) for an introduction to modern detectors of the YOLO family and their relationship with geometric descriptors such as *bounding boxes* extracted by segmentation.
- Singh (2024) for the construction of complex mazes based on Hamiltonian cycles over quasicrystalline mosaics, used as an application example of the Geodesic Distance Transform and pathfinding algorithms.
- Zampirolli (2025) for the implementation of morphological operators, geodesic transforms, and maze solving by distance propagation in restricted domains.

4.9 💻 Practical Part with Programming Exercises

This list transforms the concepts from Chapter 4 into a practical track on segmentation and mathematical morphology. The programming exercises begin with thresholding and advance to labeling and component descriptors, always using small matrices so that each pixel can be verified by hand.

> ! Common rule for morphological programming exercises
>
> In neighborhood operations, **do not apply padding**. For each pixel, evaluate only the positions of the structuring element that fall within the image domain. This is the same idea behind the didactic implementations in `morph.py`, such as `mm::dil0`, `mm::ero0`, `mm::dil1`, and `mm::label0`: the neighborhood is clipped by the valid domain of the image.

🎯 Objective of this Notebook

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

Download

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

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # C++ track build artifacts (.cpp, binary, PNGs)

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")

# The kernel is Python even in the C++ track: `mm` (morph.py) is used by the
# simulators, by the display of the figures the C++ binary generates and by the
# mm::Image state between cells. cpp=True also downloads the compiled track
# (morph.hpp + stb_image*.h), used in the #include of the %%writefile *.cpp cells.
import config
config.setup(testsuite=True, cpp=True)
from morph import mm
from testsuite import TestSuite
```

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

Executing Tests

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

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

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

4.9.1 EP04_01 Global Thresholding with a Fixed Threshold

In **document scanners** and **barcode reading systems**, the first processing step is always to separate what is "object" (ink, text, bars) from what is "background" (paper, packaging). **Global thresholding** does exactly this: it compares each pixel to a single threshold T and decides, in real time, whether it belongs to the light class or the dark class. It is the simplest segmentation operator—and yet, it underlies a large portion of industrial visual inspection *pipelines*. See Figure 4.30 for a simulation of this EP.

4.9.1.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Threshold:** Read the integer T (decision threshold).
3. **Data:** Read the integer values of the original matrix row by row.
4. **Mapping:** For each pixel p, compute the new value using the equation:

$$p' = \begin{cases} 255, & \text{if } p > T \\ 0, & \text{if } p \leq T \end{cases}$$

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

4.9.1.2 📌 Computational Constraints

- **Binarization:** The output contains **only** the values 0 or 255.
- **Strict comparison:** The criterion uses $> T$ (pixels equal to T become background).
- **Type:** The final result must be an integer.
- **Note:** This EP follows the OpenCV convention (`cv2.THRESL_BINARY`): only pixels with value **greater than** T become white (`255`); pixels with value **equal to** T remain black (`0`).

4.9.1.3 🧠 Theoretical Foundation

Parameter	Type	Visual Impact
T **small**	Integer	Most pixels become white
T **large**	Integer	Most pixels become black
T **well-chosen**	Integer	Clearly separates object and background

4.9.1.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer T.
- Following lines: Integer elements of the original matrix.

Output:

- Binarized matrix with L rows and C columns, values 0 or 255 separated by spaces.

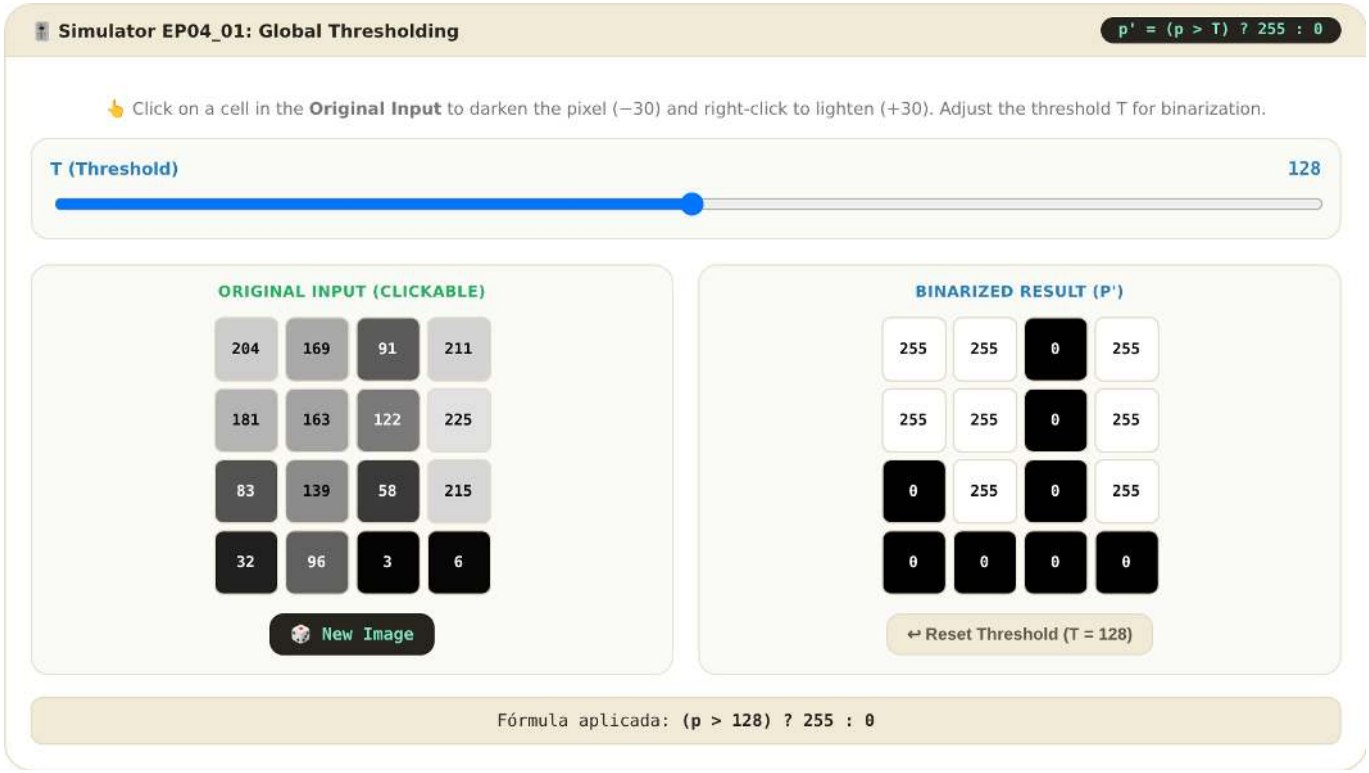
4.9.1.5 📌 Examples

Input	Output	Observation
2 4 100 0 99 100 180 255 30 120 80	0 0 0 255 255 0 255 0	$T = 100$: only pixels with value greater than 100 become white; therefore, 99 and 100 become black.
1 3 0 0 50 255	0 255 255	$T = 0$: only pixels with value strictly greater than 0 become white.

```
%%writefile EP04_01.cpp
// your solution
```

```
Overwriting EP04_01.cpp
```

```
TestSuite("EP04_01.cpp").run()
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-ep0401-limiar.html

Figure 4.30: EP04_01 Simulator: Global Thresholding by Fixed Threshold (p' = (p > T) ? 255 : 0)

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

4.9.2 EP04_02 📊 Otsu's Automatic Thresholding

Manually choosing the threshold T works when illumination is stable, but in **digital microscopy** and **blood smear inspection**, each sample has different contrast — a fixed threshold would fail from image to image. **Otsu's method** solves this by autonomously finding the threshold that **maximizes the statistical separation** between the two pixel classes, making segmentation automatic and adaptive. See Figure 4.31 for a simulation of this EP.

4.9.2.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Data:** Read the integer values of the original matrix row by row.
3. **Histogram:** Build the histogram $h[i]$, $i = 0, \ldots, 255$, counting how many pixels have value i.
4. **Threshold search:** For each candidate T from 1 to 255, compute the **between-class variance**:

$$\sigma_B^2(T) = \frac{n_0 \cdot n_1}{N^2} (m_0 - m_1)^2$$

   where n_0, n_1 are the numbers of pixels with values $< T$ and $\geq T$, m_0, m_1 are their means, and $N = L \times C$.
5. **Selection:** The optimal threshold T^* is the one that maximizes $\sigma_B^2(T)$ (in case of a tie, keep the **first** one found).
6. Application: Binarize the image using T*, applying:

$$p' = \begin{cases} 255, & \text{if } p > T^* \\ 0, & \text{if } p \leq T^* \end{cases}$$

4.9.2.2 📌 Computational Constraints

- **Valid candidates:** Ignore T that leaves $n_0 = 0$ or $n_1 = 0$ (empty class).
- **Tie-breaking:** Always keep the **first** T that reached the maximum value of σ_B^2.
- **Type:** T^* and the output matrix must be integers.
- **OpenCV convention:** The binarization follows `cv2.THRESL_BINARY`; pixels with value exactly equal to T^* become black.

4.9.2.3 🧠 Theoretical Background

Concept	Meaning	Impact
High $\sigma_B^2(T)$	Classes well separated at T	T is a good threshold candidate
Bimodal histogram	Two distinct "peaks"	Otsu finds the valley between them
Unimodal histogram	A single "peak"	Otsu still chooses *some* T, but the segmentation is unreliable

4.9.2.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Following lines: Integer elements of the original matrix.

Output:

- Binarized matrix with L rows and C columns, values 0 or 255.

4.9.2.5 📌 Examples

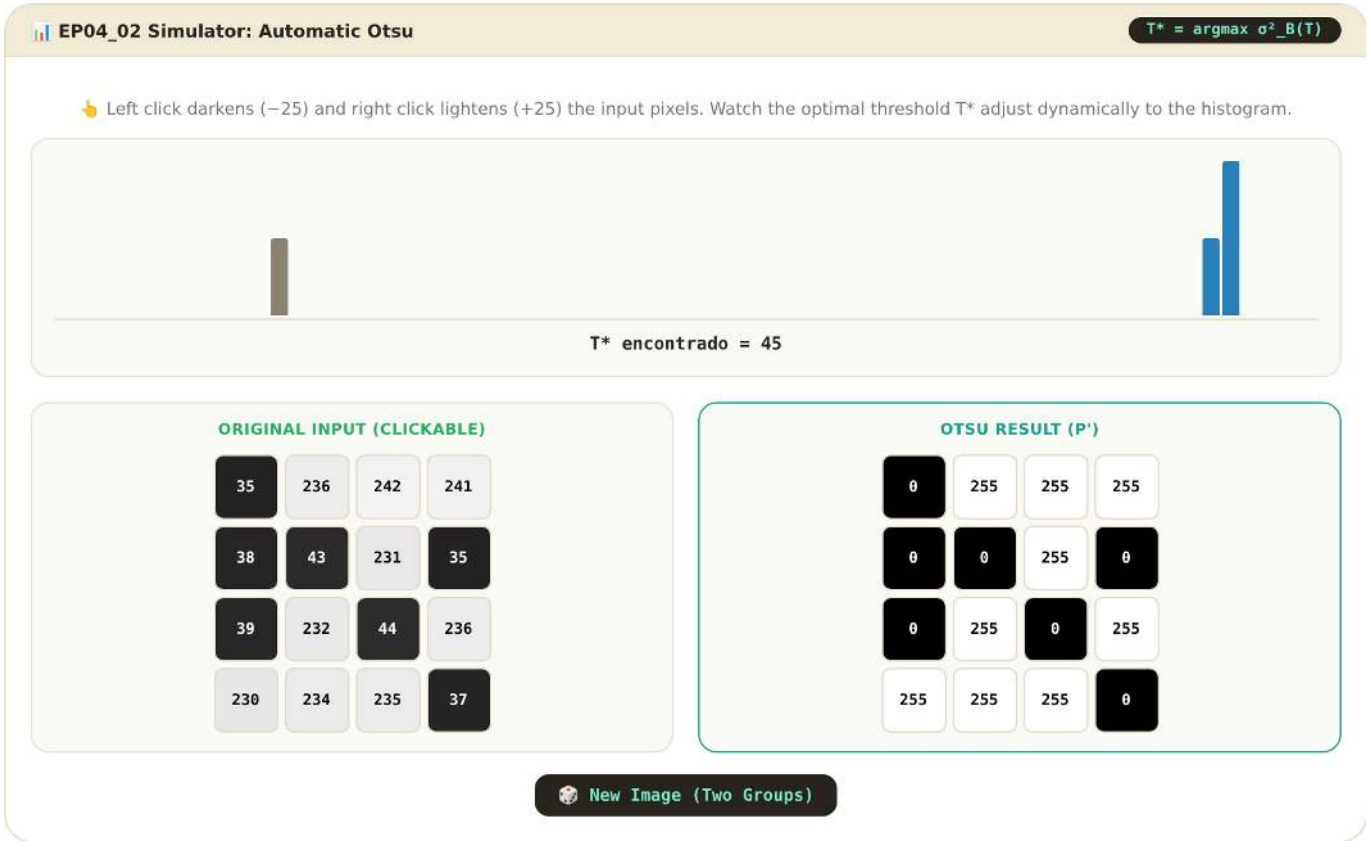
Input	Output	Observation
4 4 12 12 12 200 12 12 200 200 12 200 200 200 200 200 200 200	0 0 0 255 0 0 255 255 0 255 255 255 255 255 255 255	Clear bimodal histogram: 12 and 200
1 2 10 250	0 250	Only two values: T^* falls on the largest one

```
%%writefile EP04_02.cpp
// your solution
```

```
Overwriting EP04_02.cpp
```

```
TestSuite("EP04_02.cpp").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-ep0402-otsu.html

Figure 4.31: EP04_02 Simulator: Automatic Otsu Thresholding (T* = argmax ²_B(T))

4.9.3 EP04_03 🌱 Flat Binary Dilation (mm.dil0)

In **particle microscopy** and in **OCR of worn license plates**, thin or discontinuous traces need to be "thickened" for recognition to work. **Morphological dilation** does exactly that: it expands bright regions using a structuring element B — the same operation implemented in `morph.py` as `mm::dil0(f, B)`, used when B is **flat** (without weights, only 0/1). See Figure 4.32 for a simulation of this EP.

4.9.3.1 📋 Implementation Guidelines

1. **Image dimensions:** Read the integers L (rows) and C (columns) from f.
2. **Dimensions of B:** Read the integers L_B (rows) and C_B (columns) of the structuring element.
3. **Structuring element:** Read the matrix B with values 0 or 1, row by row.
4. **Data:** Read the matrix f (the original image), row by row.
5. **Reflection:** Construct B_{ref}, the version of B reflected by 180° (rows and columns reversed) — exactly as `mm::dil0` does internally.
6. **Neighborhood without padding:** For each pixel (y, x), traverse the positions (by, bx) of B_{ref} centered at (y, x), using the offset

$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\frac{L_B}{2} + 0{,}5, \quad o_x = -\frac{C_B}{2} + 0{,}5$$

   Discard every (v_y, v_x) outside $[0, L) \times [0, C)$ — **do not pad with zeros**.
7. **Mapping:** Compute each output pixel as the **maximum** between $f(y, x)$ and all valid $f(v_y, v_x)$ whose corresponding position in B_{ref} is 1:

$$g(y,x) = \max\left(f(y,x), \max_{\substack{(v_y,v_x)\ \text{valid} \\ B_{ref}(by,bx)=1}} f(v_y, v_x)\right)$$

8. **Output:** Display the matrix g with dimensions $L \times C$.

4.9.3.2 📌 Computational Constraints

- **No padding:** Never invent neighbors outside the image; use only those that actually exist.
- **Mandatory reflection:** B must be reflected before being applied (this is what distinguishes `mm::dil0` from a simple maximum search).
- **Edge robustness:** If no valid position of $B_{ref} = 1$ falls within the domain for a given pixel, it **maintains its original value**.

4.9.3.3 🧠 Theoretical Foundation

Concept	Meaning	Visual Impact
Dilation	$g \geq f$ always (extensive)	Bright regions grow, dark holes shrink
Larger B	Wider neighborhood	More aggressive growth
Reflection of B	$B_{ref}(y, x) = B(-y, -x)$	Ensures the formal Minkowski definition of dilation

4.9.3.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer L_B.
- Line 4: Integer C_B.
- Next L_B lines: integer elements (0 or 1) of the matrix B.
- Next L lines: integer elements of the matrix f.

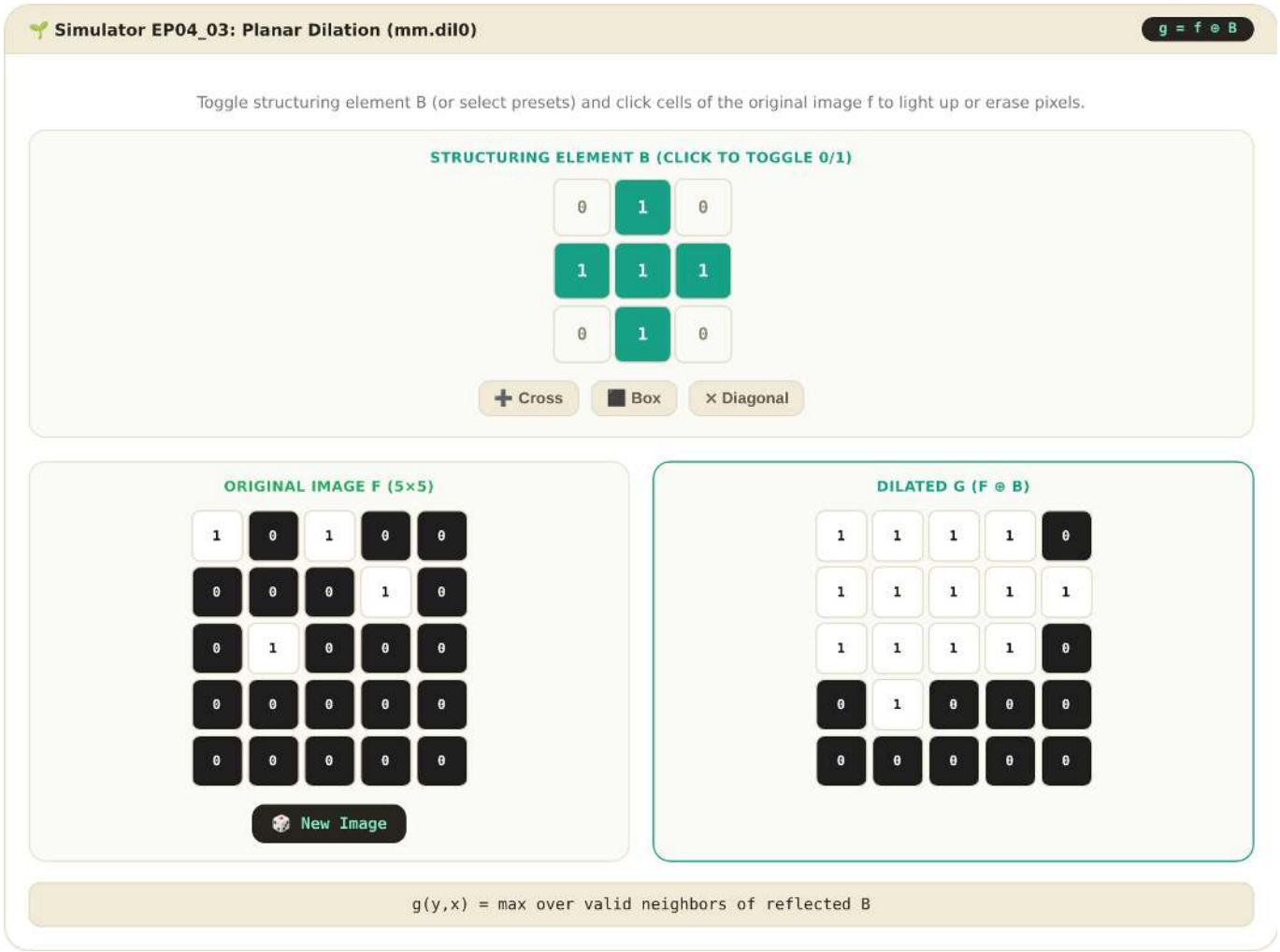
Output:

- Matrix g in L rows and C columns, integer values separated by spaces.

4.9.3.5 📌 Examples

Input	Output	Remark
3 3 3 3 0 1 0 1 1 1 0 1 0 0 0 0 0 9 0 0 0 0	0 9 0 9 9 9 0 9 0	Symmetric cross B: isolated point expands into a cross
1 4 1 3 1 1 1 10 200 5 80	200 200 200 80	Horizontal B: each pixel “pulls” the maximum of row neighbors

```
%%writefile EP04_03.cpp
// your solution
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-ep0403-dilatacao.html

Figure 4.32: Simulator EP04_03: Binary Dilation (g = f B)

```
Overwriting EP04_03.cpp
```

```
TestSuite("EP04_03.cpp").run()
```

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

4.9.4 EP04_04 🪨 Binary Erosion by Flat Structuring Element (mm.ero0)

If dilation thickens, **erosion** thins. In **cell counting systems**, it is used to **separate touching cells**: by "eating away" the borders of each region, thin connections between objects disappear even before any counting is performed. In `morph.py`, this is the operation `mm::ero0(f, B)` — the exact **dual** of dilation, and the only one of the two that does **not** reflect the structuring element. See Figure 4.33 for a simulation of this exercise.

4.9.4.1 📋 Implementation Guidelines

1. **Image dimensions:** Read the integers L (rows) and C (columns) from f.
2. **Dimensions of B:** Read the integers L_B (rows) and C_B (columns) of the structuring element.
3. **Structuring element:** Read the matrix B with values 0 or 1, row by row.
4. **Data:** Read the matrix f (the original image), row by row.
5. **Neighborhood without padding (no reflection!):** For each pixel (y, x), traverse the positions (by, bx) of B **in the original order** (without reflecting), using the same offset as in EP04_03:

$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\tfrac{L_B}{2} + 0{,}5, \quad o_x = -\tfrac{C_B}{2} + 0{,}5$$

Discard every (v_y, v_x) outside $[0, L) \times [0, C)$. 6. **Mapping:** Compute each output pixel as the **minimum** between $f(y, x)$ and all valid $f(v_y, v_x)$ whose corresponding position in B equals 1:

$$g(y,x) = \min\left(f(y,x), \min_{\substack{(v_y,v_x)\ \text{valid} \\ B(by,bx)=1}} f(v_y, v_x)\right)$$

7. **Output:** Display the matrix g with dimensions $L \times C$.

4.9.4.2 📌 Computational Constraints

- **No reflection:** Unlike dilation, B is used **exactly as read** — reflecting it here would be a serious conceptual error.
- **No padding:** Neighbors outside the image are simply ignored, never treated as 0.
- **Edge robustness:** If no valid position of $B = 1$ falls within the domain, the pixel retains its original value.

4.9.4.3 🧠 Theoretical Background

Concept	Meaning	Visual Impact
Erosion	$g \leq f$ always (anti-extensive)	Bright regions shrink, point noise disappears
Duality	$\text{ero}(f, B) = -\text{dil}(-f, B_{ref})$	Erosion and dilation are mathematical "mirrors"
Larger B	More aggressive erosion	Thin objects disappear completely

4.9.4.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer L_B.
- Line 4: Integer C_B.
- Next L_B lines: integer elements (0 or 1) of matrix B.
- Next L lines: integer elements of matrix f.

Output:

- Matrix g in L rows and C columns, integer values separated by spaces.

4.9.4.5 📌 Examples

Input	Output	Observation
3 3 3 3 0 1 0 1 1 1 0 1 0 9 9 9 9 0 9 9 9 9	9 0 9 0 0 0 9 0 9	The central “hole” (0) propagates in a cross pattern
1 4 1 3 1 1 1 10 200 5 80	10 5 5 80	Horizontal B: each pixel “pulls” the minimum of its row neighbors

```
%%writefile EP04_04.cpp
// your solution
```

```
Overwriting EP04_04.cpp
```

```
TestSuite("EP04_04.cpp").run()
```

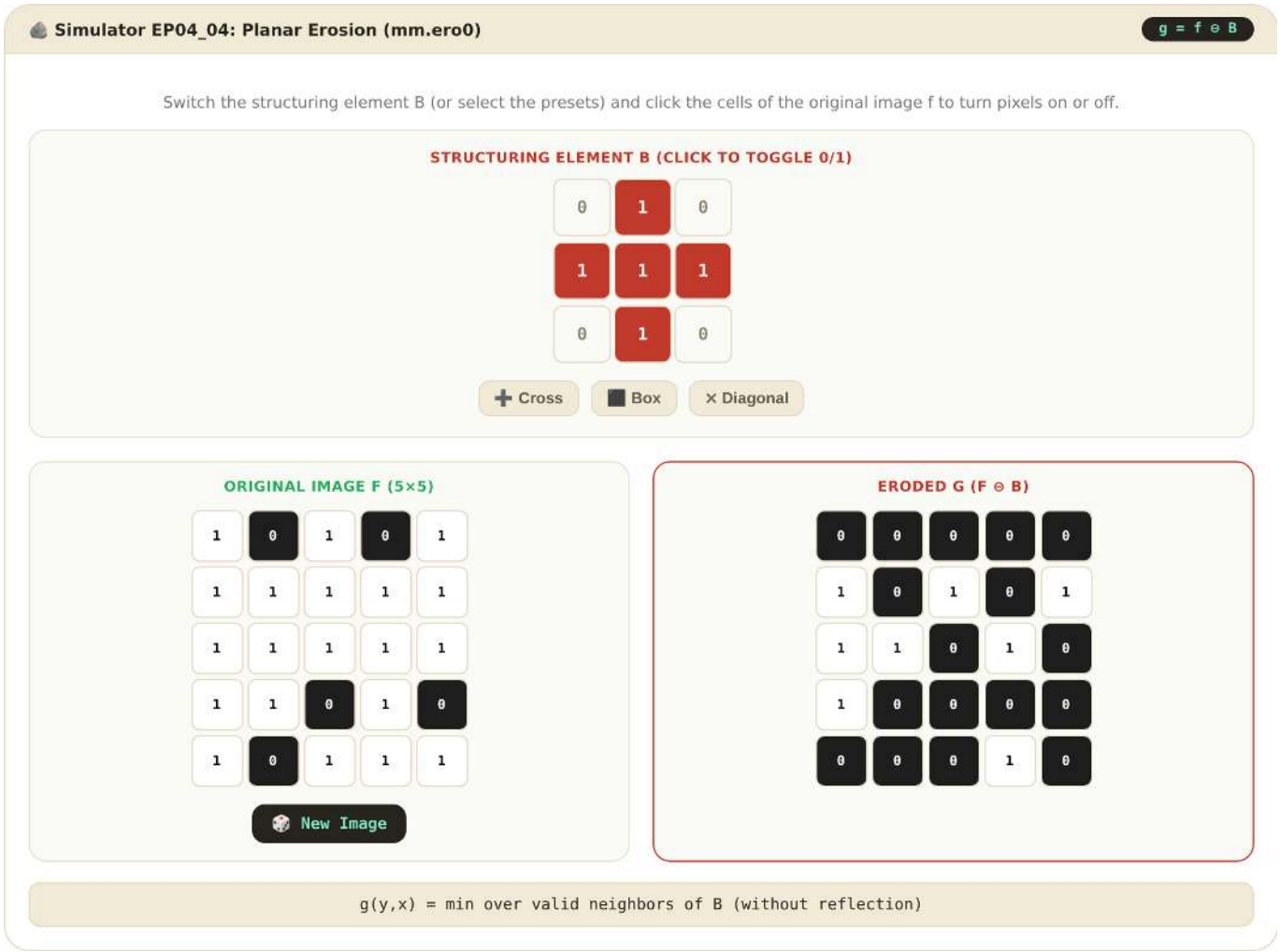
```
<IPython.core.display.HTML object>
```

4.9.5 EP04_05 🧹 Morphological Opening (Noise Removal)

Images captured by **low-cost sensors**, such as those on agricultural drones, often come dotted with small noise points—isolated pixels that represent nothing real. Applying erosion followed by dilation with the **same** structuring element produces the **opening**: it “cleans” points and thin protrusions, but returns the main object to nearly its original size. This is the classic combination used in **satellite image preprocessing** before any planted-area counting. See Figure 4.34 for a simulation of this EP.

4.9.5.1 📋 Implementation Guidelines

1. **Image dimensions:** Read integers L (rows) and C (columns) from f.
2. **Dimensions of B:** Read integers L_B (rows) and C_B (columns) of the structuring element.
3. **Structuring element:** Read matrix B with values 0 or 1, row by row.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-ep0404-erosao.html

Figure 4.33: EP04_04 Simulator: Planar Binary Erosion (g = f B)

4. **Data:** Read binary matrix f (values 0 or 1), row by row.
5. **Erosion:** Compute $e = f \ominus B$, using exactly the algorithm from EP04_04 (without reflecting B, without padding).
6. **Dilation:** Compute $g = e \oplus B$, using exactly the algorithm from EP04_03 (reflecting B, without padding)—but now applied to e, not to f.
7. **Output:** Display the resulting matrix g (the **opening** of f by B) with dimensions $L \times C$.

4.9.5.2 📌 Computational Constraints

- **Fixed order:** It is **always** erosion first, then dilation—the reverse order defines another operator (closing, from the next EP).
- **Same B:** The structuring element used in erosion and dilation must be identical.
- **No padding in either step.**

4.9.5.3 🧠 Theoretical Foundation

Concept	Meaning	Visual Impact
Anti-extensivity	$g \subseteq f$ always	The opening never creates a new pixel, only removes
Idempotence	$\text{opening}(\text{opening}(f)) = \text{opening}(f)$	Applying it again changes nothing further
Isolated points	Smaller than B	Are completely eliminated
Object core	Larger than B	Is recovered almost intact by the final dilation

4.9.5.4 📦 Input and Output Specification (VPL)

Input:

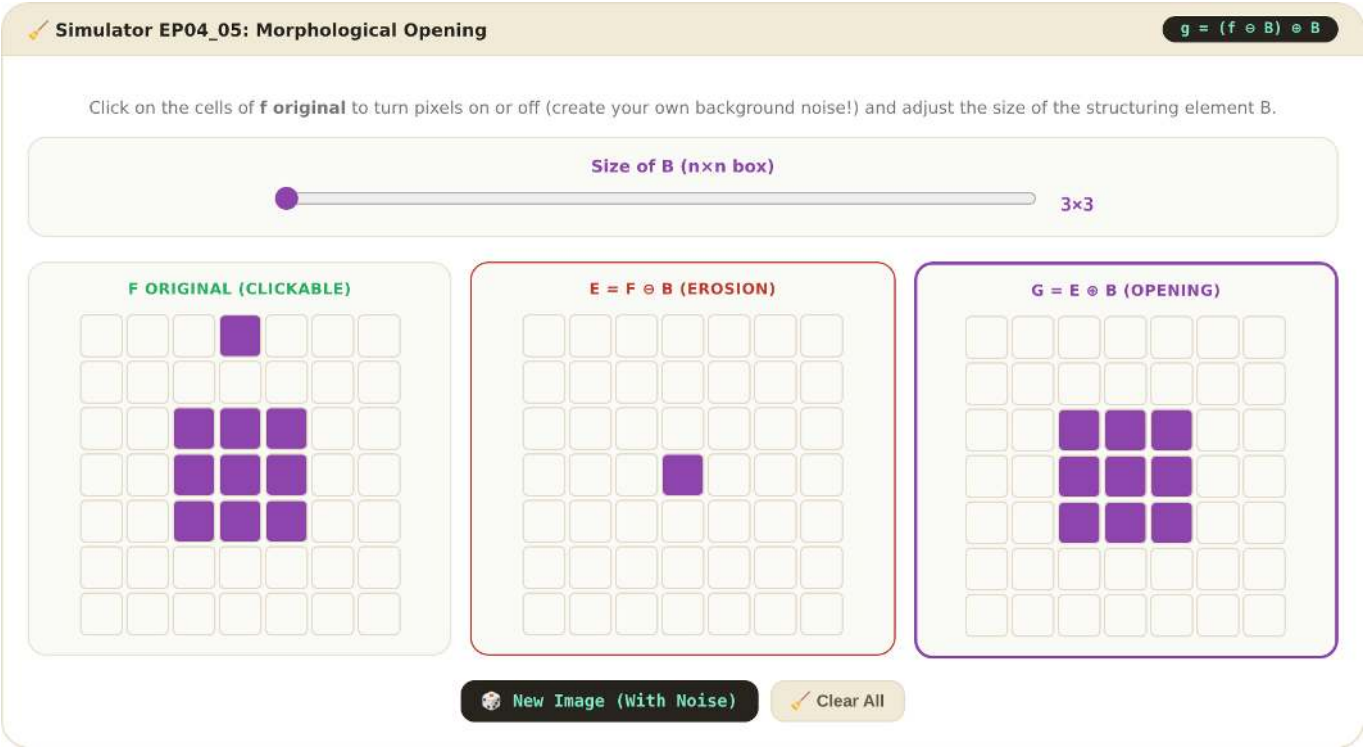
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer L_B.
- Line 4: Integer C_B.
- Next L_B lines: integer elements (0 or 1) of matrix B.
- Next L lines: integer elements (0 or 1) of matrix f.

Output:

- Resulting matrix with L rows and C columns, values 0 or 1.

4.9.5.5 📌 Examples

Input	Output	Observation
7 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	Isolated points and the thin protrusion disappear; the central square survives



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-ep0405-abertura.html

Figure 4.34: EP04_05 Simulator: Morphological Opening (g = (f B) B)

```
%%writefile EP04_05.cpp
// your solution
```

```
Overwriting EP04_05.cpp
```

```
TestSuite("EP04_05.cpp").run()
```

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

4.9.6 EP04_06 🧩 Morphological Closing (Filling of Gaps)

In **fingerprint digitalization**, skin ridges sometimes become interrupted by dirt or dryness, creating small gaps in the continuous curve that should exist. **Closing** — dilation followed by erosion with the same structuring element — is the dual operator of opening: it **fills small holes and narrow indentations**, without significantly altering the external contour of the object. It is the standard step before extracting the skeleton of a fingerprint. See Figure 4.35 for a simulation of this EP.

4.9.6.1 📋 Implementation Guidelines

1. **Image dimensions:** Read the integers L (rows) and C (columns) from f.
2. **Dimensions of B:** Read the integers L_B (rows) and C_B (columns) of the structuring element.
3. **Structuring element:** Read the matrix B with values 0 or 1, row by row.
4. **Data:** Read the binary matrix f (values 0 or 1), row by row.
5. **Dilation:** Compute $d = f \oplus B$, using exactly the algorithm from EP04_03 (reflecting B, without padding).
6. **Erosion:** Compute $g = d \ominus B$, using exactly the algorithm from EP04_04 (without reflecting B, without padding) — now applied to d, not to f.
7. **Output:** Display the resulting matrix g (the **closing** of f by B) with dimensions $L \times C$.

4.9.6.2 📌 Computational Constraints

- **Fixed order:** It is **always** dilation first, then erosion — the reverse order corresponds to the opening from EP04_05.
- **Same B:** The structuring element used in the dilation and in the erosion must be identical.
- **No padding in either of the two stages.**

4.9.6.3 🧠 Theoretical Foundation

Concept	Meaning	Visual Impact
Extensivity	$g \supseteq f$ always	Closing never removes a pixel, only adds
Idempotence	$\text{close}(\text{close}(f)) = \text{close}(f)$	Applying it again changes nothing further
Small holes	Smaller than B	They are completely filled
Duality	$\text{close}(f) = \overline{\text{open}(\bar{f})}$	It is the opening applied to the "negative" of the image

4.9.6.4 📦 Input and Output Specification (VPL)

Input:

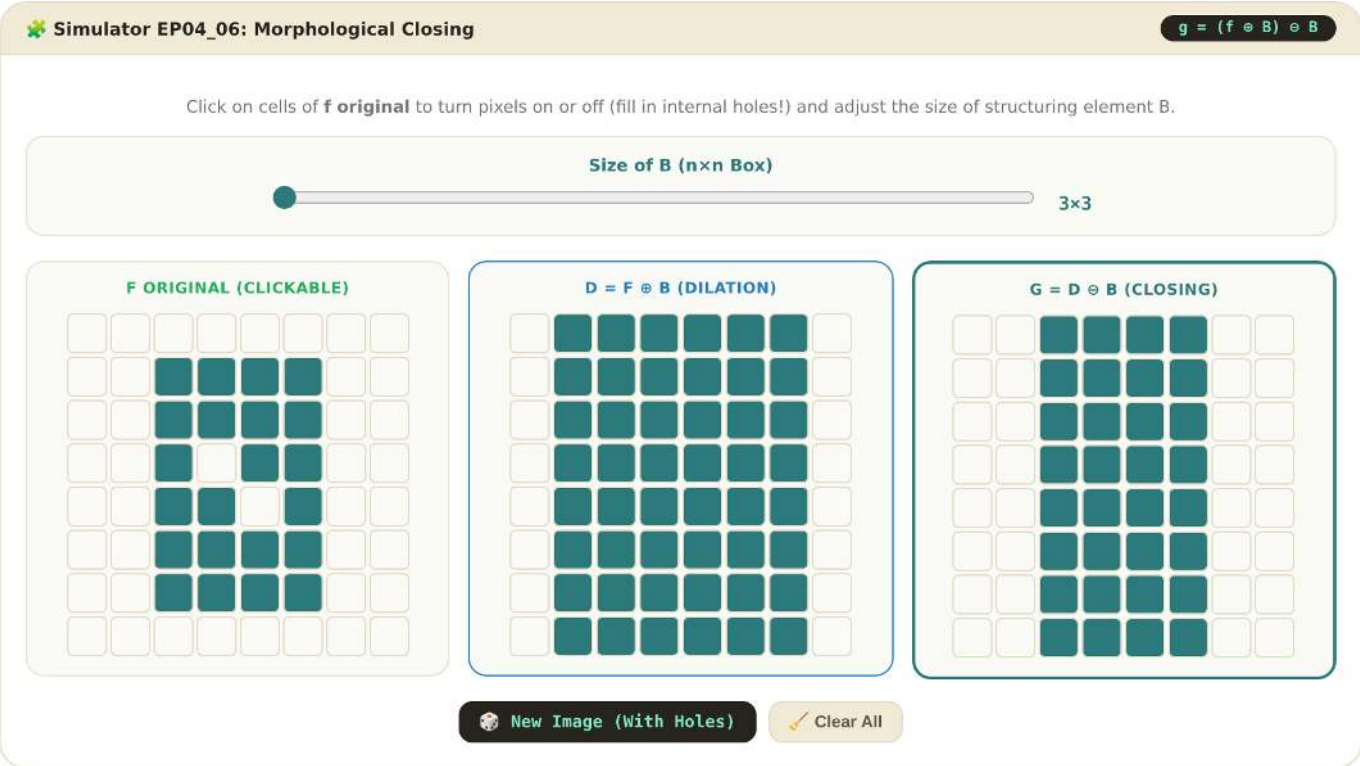
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer L_B.
- Line 4: Integer C_B.
- Next L_B lines: integer elements (0 or 1) of matrix B.
- Next L lines: integer elements (0 or 1) of matrix f.

Output:

- Resulting matrix in L rows and C columns, values 0 or 1.

4.9.6.5 📌 Examples

Input	Output	Observation
8 8 3 3 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0	0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0	The two non-adjacent internal holes are completely filled



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-ep0406-fechamento.html

Figure 4.35: EP04_06 Simulator: Morphological Closing (g = (f B) B)

```
%%writefile EP04_06.cpp
// your solution
```

```
Overwriting EP04_06.cpp
```

```
TestSuite("EP04_06.cpp").run()
```

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

4.9.7 EP04_07 Weighted Dilation and Erosion (mm.dil1 / mm.ero1)

So far, the structuring element only indicated "this neighbor counts" or "does not count" — but in **digital elevation models** (used in GIS and urban drainage planning), each neighbor should have a **different weight** depending on the distance or the direction of the terrain. The **weighted** versions of dilation and erosion, implemented in `morph.py` as `mm::dil1(f, b)` and `mm::ero1(f, b)`, sum (or subtract) the weight of each neighbor before taking the maximum (or minimum) — generalizing everything done in the previous EPs. See Figure 4.36 for a simulation of this EP.

4.9.7.1 📋 Implementation Guidelines

1. **Image dimensions:** Read the integers L (rows) and C (columns) from f.
2. **Dimensions of b:** Read the integers L_B (rows) and C_B (columns) of the weighted structuring element.
3. **Weights:** Read the matrix b of **integer** weights (which may be negative, zero, or positive), row by row.
4. **Data:** Read the matrix f (the original image), row by row.
5. **Neighborhood without padding:** For each pixel (y, x), iterate over **all** positions (by, bx) of b (not only where it would equal 1 — here **every** weight participates), using the same offset as in previous EPs:
$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\tfrac{L_B}{2} + 0{,}5, \quad o_x = -\tfrac{C_B}{2} + 0{,}5$$
Discard every (v_y, v_x) outside $[0, L) \times [0, C)$.
6. **Weighted dilation:** Compute
$$g_{dil}(y, x) = \max\left(f(y, x), \max_{(v_y, v_x)\ \text{valid}} \left(f(v_y, v_x) + b(by, bx)\right)\right)$$
7. **Weighted erosion:** Compute, **using the same b and without reflection**:
$$g_{ero}(y, x) = \min\left(f(y, x), \min_{(v_y, v_x)\ \text{valid}} \left(f(v_y, v_x) - b(by, bx)\right)\right)$$
8. **Output:** Display **first** the complete matrix g_{dil}, and **then** the complete matrix g_{ero}.

4.9.7.2 📌 Computational Constraints

- **Neither reflects** b — the weighted version does not use reflection, even in dilation (unlike `mm::dil0`).
- **All weights participate:** There is no "$B = 1$" filter here; even weight 0 is included in the computation.
- **No padding:** neighbors outside the image are ignored, never virtually filled.
- **Type:** The output may contain negative values or values greater than 255 — there is **no** *clipping* in this EP.
- **Hint:** To remove overflow messages when exceeding uint8 limits, include at the beginning of the code:

```
import warnings
warnings.filterwarnings("ignore")
```

4.9.7.3 🧠 Theoretical Foundation

Concept	Meaning	Visual Impact
Positive weight	"Pulls" the neighbor's value upward in dilation	Simulates terrain rising in that direction
Negative weight	Reduces the neighbor's contribution	Simulates distance or directional attenuation
Weighted duality	$\text{ero1}(f, b) = -\text{dil1}(-f, b)$	The symmetry between the two operations is maintained even with weights

4.9.7.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer L_B.
- Line 4: Integer C_B.
- Next L_B lines: integer elements (may be negative) of the matrix b.
- Next L lines: integer elements of the matrix f.

Output:

- First, the matrix g_{dil} in L rows and C columns.
- Then, the matrix g_{ero} in L rows and C columns.

4.9.7.5 📌 Examples

Input	Output	Observation
3 3 3 3 0 1 0 1 2 1 0 1 0 10 20 30 40 50 60 70 80 90	50 60 61 80 90 91 81 91 92 8 9 19 9 10 20 39 40 50	Central weight 2 accelerates growth in dilation and shrinkage in erosion

```
%%writefile EP04_07.cpp
// your solution
```

```
Overwriting EP04_07.cpp
```

```
TestSuite("EP04_07.cpp").run()
```

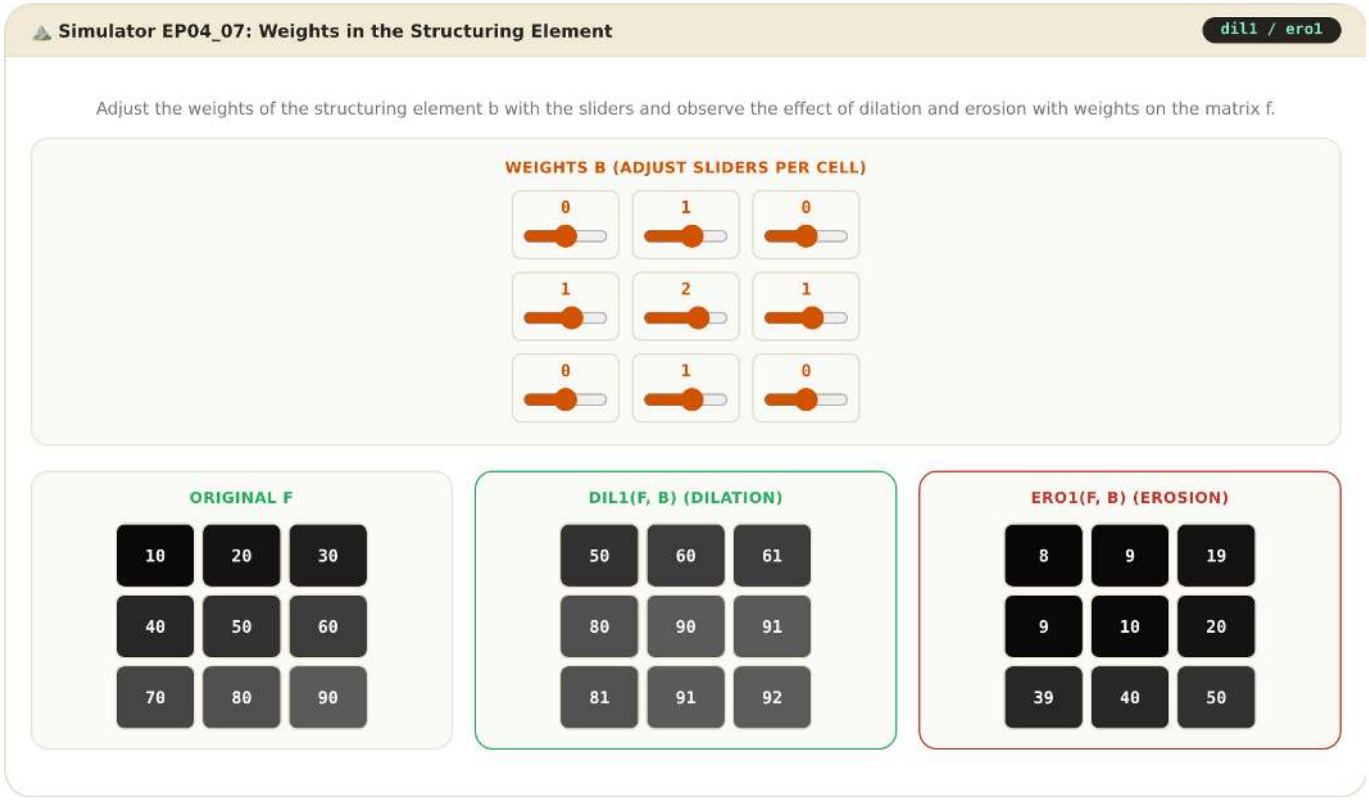
```
<IPython.core.display.HTML object>
```

4.9.8 EP04_08 🌋 Morphological Gradient, Top-hat, and Black-hat

In **automatic inspection of printed circuit boards**, three questions arise all the time: where are the **edges** of the components? Which **small bright details** (such as solder points) stand out from the background? What **dark recesses** (such as cracks) does the background conceal? A single erosion/dilation pair answers all three: the **morphological gradient** highlights contours, the **top-hat** reveals narrow peaks, and the **black-hat** reveals narrow valleys — three tools, one single neighborhood. See Figure 4.37 for a simulation of this EP.

4.9.8.1 📋 Implementation Guidelines

1. **Image dimensions:** Read the integers L (rows) and C (columns) from f.
2. **Dimensions of B:** Read the integers L_B (rows) and C_B (columns) of the structuring element.
3. **Structuring element:** Read the matrix B with values 0 or 1, row by row.
4. **Data:** Read the matrix f (the original image, in grayscale), row by row.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-ep0407-pesos.html

Figure 4.36: EP04_07 Simulator: Dilation and Erosion with Weights (mm.dil1 / mm.ero1)

5. **Basic operators:** Compute, exactly as in EPs 04_03 through 04_06:
   - $d = f \oplus B$ (dilation),
   - $e = f \ominus B$ (erosion),
   - opening $= e \oplus B$,
   - closing $= d \ominus B$.
6. **Morphological gradient:** $\text{grad}(y, x) = d(y, x) - e(y, x)$.
7. **Top-hat:** $\text{tophat}(y, x) = f(y, x) - \text{opening}(y, x)$.
8. **Black-hat:** $\text{blackhat}(y, x) = \text{closing}(y, x) - f(y, x)$.
9. **Output:** Display, **in this order**, the three complete matrices: gradient, top-hat, black-hat.

4.9.8.2 📌 Computational Constraints

- **No padding at any intermediate stage** — dilation, erosion, opening, and closing follow the same neighborhood rules as in the previous EPs.
- **No clipping:** the three outputs may contain any integer value (the gradient is always ≥ 0, but top-hat and black-hat may also be so).
- **Reuse:** d and e must be computed **only once** and reused to assemble opening, closing, and gradient.

4.9.8.3 🧠 Theoretical Foundation

Operator	Formula	What it reveals
Gradient	$d - e$	Edges: zero in flat regions, high at transitions
Top-hat	$f - \text{opening}(f)$	**Bright, thin** elements, smaller than B
Black-hat	$\text{closing}(f) - f$	**Dark, thin** elements, smaller than B

4.9.8.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer L_B.
- Line 4: Integer C_B.
- Next L_B lines: integer elements (0 or 1) of matrix B.
- Next L lines: integer elements of matrix f.

Output:

- Gradient matrix with L rows and C columns.
- Top-hat matrix with L rows and C columns.
- Black-hat matrix with L rows and C columns.

4.9.8.5 📌 Examples

Input	Output	Observation
9 9 3 3 1 1 1 1 1 1 1 1 1 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 80 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 2 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10	(gradient: 3×3 halo of 70 around $(2, 2)$ and 3×3 halo of 8 around $(6, 6)$, rest 0) (top-hat: single 70 at $(2, 2)$, rest 0) (black-hat: single 8 at $(6, 6)$, rest 0)	Isolated peak becomes top-hat; isolated valley becomes black-hat; both appear in the gradient

```
%%writefile EP04_08.cpp
// your solution
```

```
Overwriting EP04_08.cpp
```

```
TestSuite("EP04_08.cpp").run()
```

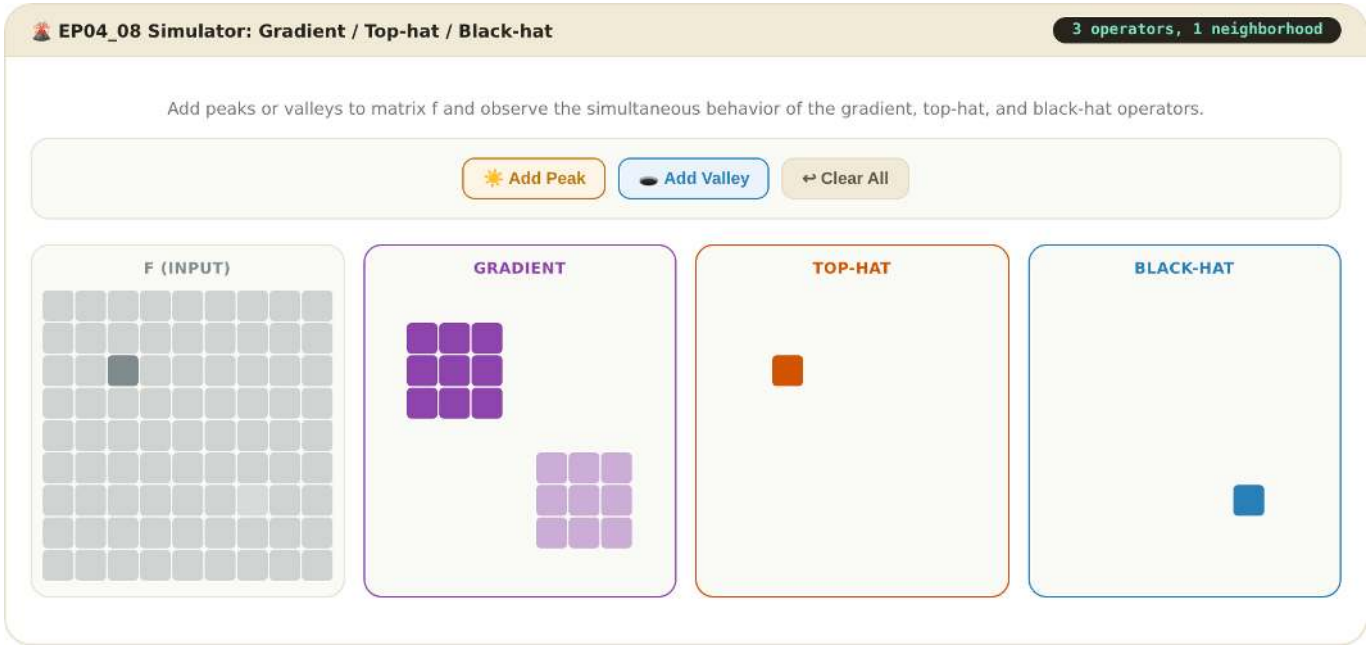
```
<IPython.core.display.HTML object>
```

4.9.9 EP04_09 Distance Transform and the Object's "Core"

In **mobile robotics**, when planning a route within a corridor, the robot wants to know not only *where* free space exists, but also **how far** each free point is from the nearest wall. The safest paths tend to pass through the corridor's "core," away from obstacles.

The **morphological distance transform** assigns to each pixel a value representing its distance to the nearest edge, according to the metric defined by the structuring element. Pixels near the edge receive low values, while more internal pixels receive higher values. The pixel with the maximum value corresponds to the most protected region of the object, often associated with its morphological center.

See Figure 4.38 for a simulation of this EP.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-ep0408-gradiente.html

Figure 4.37: EP04_08 Simulator: Morphological Gradient, Top-hat and Black-hat

4.9.9.1 📋 Implementation Guidelines

1. **Image dimensions:** read the integers L (rows) and C (columns) of image f.
2. **Dimensions of B:** read the integers L_B (rows) and C_B (columns) of the structuring element.
3. **Structuring element:** read the matrix b, containing value 0 at the center and negative values at other positions.
4. **Image:** read the binary matrix f (values 0 or 1), row by row.
5. **Preparation:** multiply the image by $L \times C$, ensuring that internal pixels have an initial value sufficiently high for distance propagation.
6. **Distance transform:** compute the distance matrix using the method `mm::dist1(f,b)`.
7. **Output:** display the matrix resulting from the distance transform.

4.9.9.2 📌 Computational Constraints

- Use the weighted erosion implementation provided by the library.
- The structuring element may contain arbitrary negative values.
- The transform must be obtained by iteratively applying weighted erosions until a fixed point is reached.

⚠️ **Crucial Note on Matrix Reading:** Since the structuring element may contain negative integer values (e.g., `-1` and `-99`), **do not use the `mm::readImg` function to read matrix b**. This function converts data to `uint8` type, causing *underflow* and corrupting negative values. Read the L_B rows of b manually using the default `int` type. Image f can continue to be read normally using `mm::readImg`.

4.9.9.3 🧠 Theoretical Foundation

Concept	Meaning	Visual Impact
dist(y, x)	Morphological distance to the nearest edge according to the metric defined by b	More internal pixels receive higher values
Maximum value	Pixel farthest from the edge	Approximates the object's morphological center
Weighted structuring element	Defines the displacement costs between neighboring pixels	Determines the distance metric used

Concept	Meaning	Visual Impact
Thin objects	Narrow regions of the object	Produce low distance values

4.9.9.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: integer L.
- Line 2: integer C.
- Line 3: integer L_B.
- Line 4: integer C_B.
- Next L_B lines: integer elements of matrix b.
- Next L lines: binary elements (0 or 1) of matrix f.

⚠️ **Implementation note:** The elements of matrix f (0 or 1) must be multiplied by **255** to generate an adequate binary image (0 and 255) before applying the Distance Transform (DT).

Output:

- Distance transform matrix with L rows and C columns.

4.9.9.5 📌 Example

Input	Output	Observation
5 9 3 3 -99 -1 -99 -1 0 -1 -99 -1 -99 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 2 2 2 2 2 1 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0	Result of the distance transform.

Note: the value `-99` acts as a practical approximation of $-\infty$, preventing propagation along diagonals. Thus, only horizontal and vertical neighbors contribute to the distance, producing the Manhattan distance.

```
%%writefile EP04_09.cpp
// your solution
```

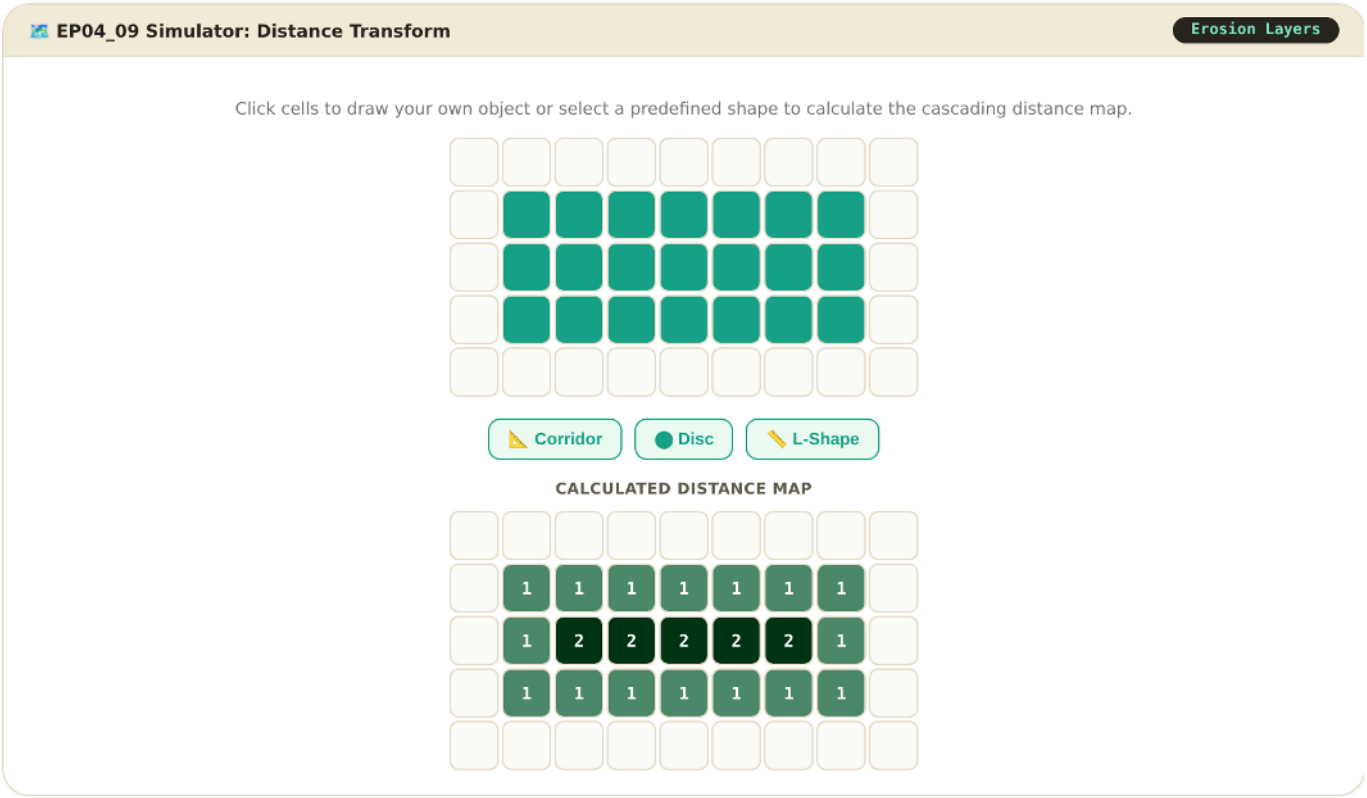
```
Overwriting EP04_09.cpp
```

```
TestSuite("EP04_09.cpp").run()
```

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

4.9.10 EP04_10 🪙 *Blob* Separation, Labeling, and Descriptors

In a **coin production line**, it is common for pieces to touch one another on the conveyor belt, forming a single connected blob in the image—a naive count would yield the wrong total. The classic solution combines



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-ep0409-distancia.html

Figure 4.38: Simulador EP04_09: Distância por Transformada (Camadas de Erosão)

morphological operations and connectivity analysis: first, an **erosion** reduces or breaks fragile connections between objects, and then **connected component labeling** separates each object into a distinct region. Finally, **geometric descriptors** (area and bounding box) summarize each detected component.

See Figure 4.39 for a simulation of this EP.

4.9.10.1 📋 Implementation Guidelines

1. **Image dimensions:** read the integers L (rows) and C (columns) from f.
2. **Dimensions of B:** read the integers L_B (rows) and C_B (columns) of the structuring element.
3. **Structuring element:** read the matrix B, containing values 0 or 1, row by row.
4. **Data:** read the binary matrix f (values 0 or 1), row by row.
5. **Separation:** compute
$$f_{ero} = f \ominus B$$
using flat binary erosion (as in EP04_04), eliminating fragile connections between objects.
6. **Labeling:** on f_{ero}, identify connected components using connectivity defined by the neighborhood B. Labeling must follow *raster* scanning: upon finding an unlabeled pixel with value 1, assign a new increasing integer label starting from 1 and propagate that label to the entire connected region.
7. **Descriptors:** for each label k, compute:
   - **Area:** number of pixels belonging to the label;
   - **Bounding box:**
$$(y_{min}, x_{min}, y_{max}, x_{max})$$
8. **Output:** display the total number of labels and then one line per label in the format:
$$k,\ \text{area},\ y_{min},\ x_{min},\ y_{max},\ x_{max}$$

4.9.10.2 📌 Computational Constraints

- Erosion must be applied before labeling.
- Connectivity is fixed and defined by the neighborhood above.
- The structuring element B does not interfere with labeling connectivity.
- No padding in any step.
- The order of labels follows first discovery in *raster* scanning.

4.9.10.3 🧠 Theoretical Background

Concept	Meaning	Impact
Thin bridge	Narrow connection between objects	Can be removed by morphological erosion
Connectivity	Defined by the set $\mathcal{N}(y, x)$	Determines which pixels belong to the same component
Area	Number of pixels per component	Direct estimate of object size
Bounding box	Spatial extent of the label	Geometric summary of the component

4.9.10.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: integer L
- Line 2: integer C
- Line 3: integer L_B
- Line 4: integer C_B
- Next L_B lines: matrix B
- Next L lines: matrix f

Output:

- Line 1: total number of labels found
- Following lines:

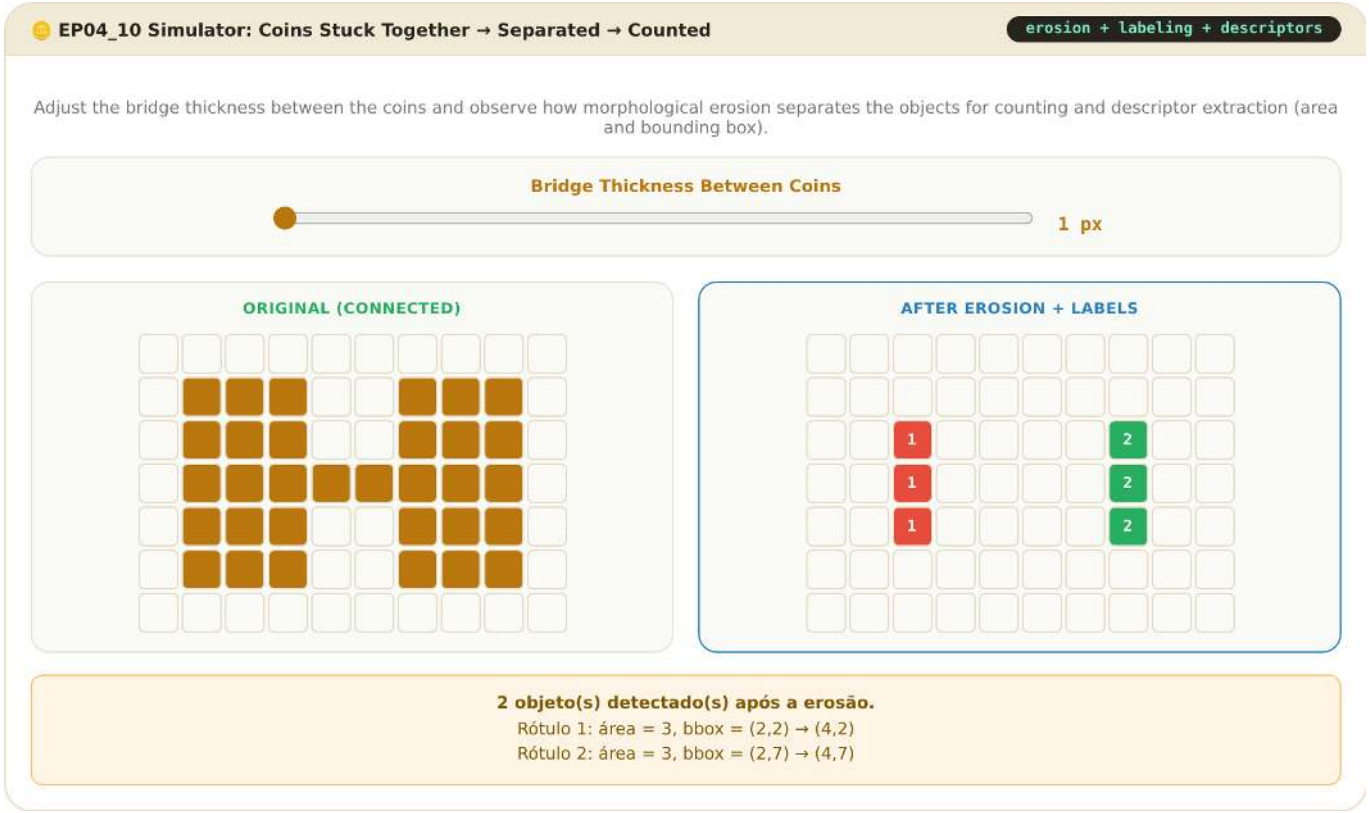
$$k,\ \text{area},\ y_{min},\ x_{min},\ y_{max},\ x_{max}$$

```
%%writefile EP04_10.cpp
// your solution
```

```
Overwriting EP04_10.cpp
```

```
TestSuite("EP04_10.cpp").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap04/sim-ep0410-rotulacao.html

Figure 4.39: Simulador EP04_10: Blob Separation, Labeling, and Descriptors

Chapter 5

Transforms and Compression

In the previous chapters, all operations were performed **in the spatial domain**, where algorithms act directly on pixel intensity values.

This chapter presents a complementary approach: the **frequency domain**, in which the image is represented by the spatial variations of intensity, rather than only by the individual pixel values.

The concept of **spatial frequency** describes how quickly intensity varies across the image. Slow variations correspond to **low frequencies**, while edges, fine details, and noise correspond to **high frequencies**.

This representation relies on the fact that any discrete digital image can be decomposed into a combination of **orthogonal functions**. The **Fourier Transform** uses a **basis of two-dimensional complex exponentials** (equivalent to sinusoids with specific orientation and frequency). Other transforms, such as the **Cosine Transform (DCT)** and the **Wavelet Transform (DWT)**, use different families of basis functions — two-dimensional cosines in the case of the DCT, and functions with compact support in the case of wavelets.

Among the main applications of this representation are:

1. **Frequency-domain filtering**, to attenuate or enhance certain frequency bands;
2. **Multiresolution analysis through wavelet transforms**, which represents structures at different scales;
3. **Image compression**, by reducing the number of coefficients needed to represent the image.

5.1 Objectives

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

- **Interpret the Fourier spectrum** of an image, distinguishing magnitude, phase, and frequency components;
- **Apply the Convolution Theorem** to perform frequency-domain filtering using the Fast Fourier Transform (FFT);
- **Design and analyze frequency-domain filters**, understanding the operation of low-pass, high-pass, and notch filters;
- **Understand multiresolution analysis via wavelet transforms** and its application to the hierarchical representation of images;
- **Describe the image compression process**, including the Discrete Cosine Transform (DCT) and coefficient quantization;
- **Select image storage formats**, such as JPEG, PNG, and WebP, according to application requirements.

5.2 Environment Setup

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # C++ track build artifacts

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")

# Python kernel even on the C++ track. cpp=True downloads morph.hpp + stb; the
# %%writefile *.cpp cells in this chapter compile WITH OpenCV
# (-DMM_USE_OPENCV + pkg-config opencv4).
import config
config.setup(cpp=True)
from morph import mm
import numpy as np
```

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

5.3 2D Discrete Fourier Transform

Fourier analysis is based on the principle that any periodic signal can be represented as a sum of sinusoidal functions with different frequencies, amplitudes, and phases. This concept also applies to digital images, allowing them to be represented in the **frequency domain** instead of the spatial domain.

Figure 5.1 illustrates this decomposition for a one-dimensional signal. In the case of an image, the Discrete Fourier Transform (DFT) converts the intensity matrix $f(x, y)$ into a set of coefficients that describes the contribution of the different spatial frequencies present in the image.

```
%%writefile tmp/fig_decomposicao_1d.cpp
#define MM_OUT "tmp/fig_decomposicao_1d.png"
//| label: fig-decomposicao-1d
//| fig-cap: "Fourier 1D decomposition: a square wave (dashed line) is approximated by the sum
of the first sinusoids (colored lines). The more terms, the better the approximation."
//| echo: false
//| output: true

#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <cmath>
#include "morph.hpp"
#include <filesystem>

int main() {
    // x = np.linspace(0, 2 * np.pi, 400)
    std::vector<double> x(400);
    for (int i = 0; i < 400; ++i) {
        x[i] = (2.0 * M_PI) * i / 399.0;
    }

    // square = np.where(x < np.pi, 1.0, -1.0)
    std::vector<double> square(400);
    for (int i = 0; i < 400; ++i) {
        square[i] = (x[i] < M_PI) ? 1.0 : -1.0;
    }
```

```
    // soma = np.zeros_like(x)
    std::vector<double> soma(400, 0.0);

    // harms = []
    std::vector<std::vector<double>> harms;

    // for n in range(1, 4):
    for (int n = 1; n <= 3; ++n) {
        // h = (4.0 / np.pi) * (1.0 / (2 * n - 1)) * np.sin((2 * n - 1) * x)
        std::vector<double> h(400);
        for (int i = 0; i < 400; ++i) {
            h[i] = (4.0 / M_PI) * (1.0 / (2.0 * n - 1.0)) * std::sin((2.0 * n - 1.0) * x[i]);
        }
        harms.push_back(h);
        // soma = soma + h
        for (int i = 0; i < 400; ++i) {
            soma[i] += h[i];
        }
    }

    // ys = [square, harms[0], harms[1], harms[2], soma]
    std::vector<std::vector<double>> ys = {square, harms[0], harms[1], harms[2], soma};

    // labels = [...]
    std::vector<std::string> labels = {"Ideal square wave", "1st harmonic", "3rd harmonic",
"5th harmonic", "Sum (first 3)"};

    // colors = [(40, 40, 40), (60, 160, 80), (180, 120, 60), (150, 80, 160), (60, 60, 220)]
    std::vector<cv::Scalar> colors = {
        cv::Scalar(40, 40, 40),
        cv::Scalar(60, 160, 80),
        cv::Scalar(180, 120, 60),
        cv::Scalar(150, 80, 160),
        cv::Scalar(60, 60, 220)
    };

    // chart = mm.lineChart(...)
    mm::Image chart = mm::lineChart(
        x, ys, labels, colors,
        "Fourier synthesis: from sines to a square wave",
        "Position", "Intensity"
    );

    // mm.show([chart], titles=["Decomposicao de Fourier 1D"], cols=1)
    std::vector<mm::Image> images = {chart};
    std::vector<std::string> titles = {"Fourier 1D decomposition"};
    mm::show(images, MM_OUT, titles, 1);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(chart, "tmp/fig_decomposicao_1d_0.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_decomposicao_1d.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_decomposicao_1d.cpp -o
tmp/fig_decomposicao_1d -lopencv_stitching -lopencv_alphamat -lopencv_aruco -lopencv_barcode
-lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_decomposicao_1d \
  && test -f "tmp/fig_decomposicao_1d.png" \
  || echo "  mm::show não gravou tmp/fig_decomposicao_1d.png"
```

```
[1] Fourier 1D decomposition
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_decomposicao_1d_0.png"),
        ],
        titles=[
            'Decomposicao de Fourier 1D',
        ],
        cols=1,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_decomposicao_1d_0.png (ver a versao Python)")
```

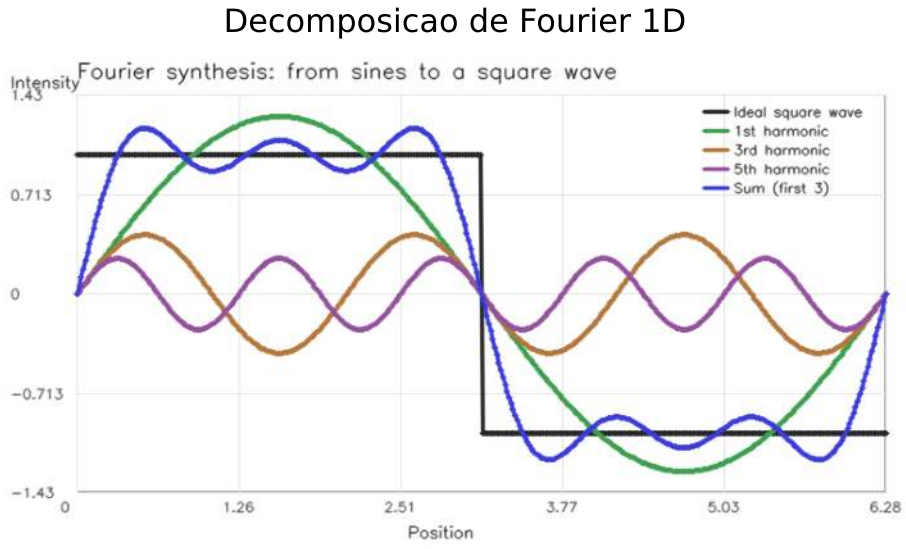


Figure 5.1: Decomposição de Fourier 1D: uma onda quadrada (linha tracejada) é aproximada pela soma das primeiras senoides (linhas coloridas). Quanto mais termos, melhor a aproximação.

5.3.1 Simulator: Reconstructing Signals with Sinusoids

Before studying two-dimensional images, the simulator in Figure 5.2 illustrates the principle of Fourier analysis for one-dimensional signals: **a waveform can be approximated by the sum of sinusoids with different frequencies and amplitudes**.

As new terms are added, the sum of the sinusoids (black curve) approaches the reference waveform (dashed). The lower plot displays the amplitude spectrum, indicating the contribution of each frequency to the reconstruction of the signal.

Activity

Explore the simulator and answer:

1. How many terms are needed to obtain a good approximation of the square wave?
2. Which of the three waveforms converges most rapidly? Justify your answer.
3. How does the amplitude spectrum change when switching from the square wave to the triangular wave?

Answers

1. How many terms are needed for a good approximation of the square wave?
With approximately 15 to 20 terms, the waveform already closely approximates the reference. However, near the discontinuities, a small oscillation remains, known as the **Gibbs phenomenon**, which does not disappear even with the addition of more terms.
2. Which waveform converges most rapidly? Why?
The **triangular wave** converges most rapidly, because the amplitudes of its harmonics decay faster than those of the square wave and the sawtooth wave. As a result, few terms already produce a good approximation.
3. How does the spectrum change between the square wave and the triangular wave?
Both contain only **odd harmonics**, but in the triangular wave, the amplitudes decrease much more rapidly. Thus, few harmonics are sufficient to reconstruct the signal with good accuracy.

5.3.2 Interpretation of the Frequency Spectrum

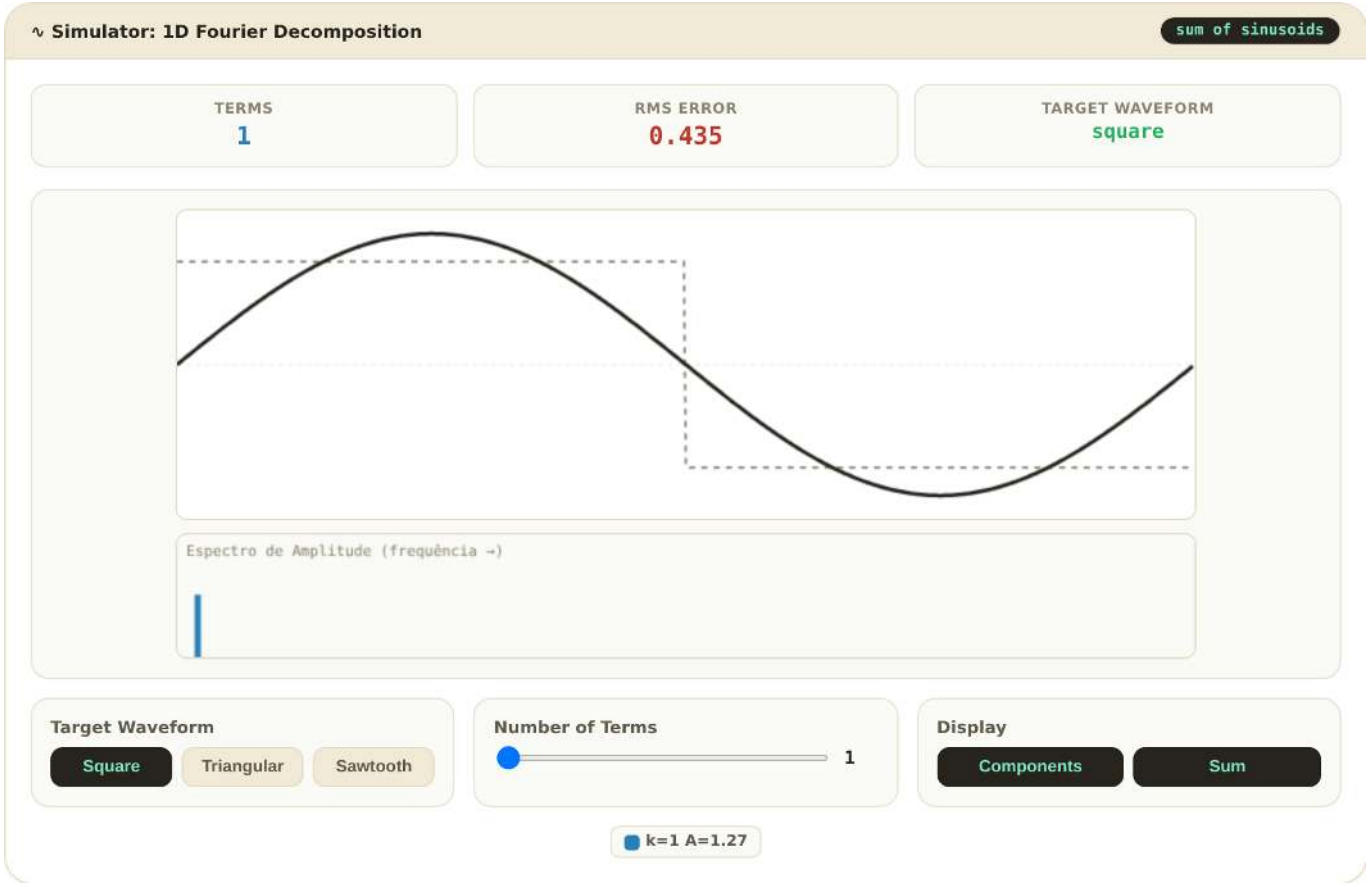
When applying the Discrete Fourier Transform (DFT) to an image and visualizing the magnitude of its coefficients (see Figure 5.5), one obtains the **magnitude spectrum**, which shows the distribution of spatial frequencies present in the image.

The coefficient located at the origin of the DFT, known as the **DC component** (*Direct Current*), corresponds to the zero frequency and represents the average intensity of the image. By convention, this coefficient is stored in the upper left corner of the spectrum. To facilitate its interpretation, the **FFT Shift** operation is applied, which shifts the DC component to the center of the image. After this shift, low frequencies are concentrated in the central region, while high frequencies are near the edges, as summarized in Table 5.1.

Table 5.1: Correspondence between the regions of the magnitude spectrum after applying the FFT Shift.

Spectrum region	Predominant components	Examples in the image
Center (low frequencies)	Slow spatial variations	Illumination, homogeneous regions, and global shapes
Intermediate region (mid frequencies)	Intermediate-scale variations	Textures and repetitive patterns
Edges (high frequencies)	Rapid spatial variations	Contours, fine details, and noise

This organization facilitates the interpretation of the spectrum and the design of filters. Attenuating low frequencies reduces global intensity variations, while attenuating high frequencies smooths the image by reducing fine details and part of the noise.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap05/sim-05-freq.html

Figure 5.2: Interactive simulator of 1D Fourier decomposition: visualization of the sum of sinusoids with different frequencies, amplitudes and phases. Add terms and observe convergence to arbitrary waveforms.

5.3.3 The Grid Experiment: Building an Image from a Single Coefficient

Before presenting the mathematical formulation of the Discrete Fourier Transform (DFT), it is useful to analyze its inverse, called the Inverse Discrete Fourier Transform (IDFT). Consider a spectrum in which all coefficients are zero, except one. An example of this construction is presented in the code of Figure 5.3 and can be explored interactively in the simulator of Figure 5.4.

The reconstructed image is a two-dimensional sinusoid. The position of the coefficient in the spectrum determines its **orientation** and its **spatial frequency**, while its magnitude and phase define, respectively, its amplitude and spatial displacement. Thus, each DFT coefficient represents a sinusoidal component, and the original image can be reconstructed by summing all these components.

```
%%writefile tmp/fig_05_grade_2d.cpp
#define MM_OUT "tmp/fig_05_grade_2d.png"
//| label: fig-05-grade-2d
//| fig-cap: "Toda frequência no espectro (ponto isolado) corresponde a uma onda senoidal 2D
rotacionada no domínio espacial."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <complex>
#include "morph.hpp"
#include <filesystem>

int main() {
    int N_grid = 100;
    cv::Mat espectro_vazio = cv::Mat::zeros(N_grid, N_grid, CV_64FC2);

    // Acendendo um único ponto (frequência) fora do centro
    int u0 = 10, v0 = 5;
    espectro_vazio.at<cv::Vec2d>(N_grid/2 - v0, N_grid/2 - u0) = cv::Vec2d(1000, 0);

    // Retornando para o domínio espacial (IDFT)
    cv::Mat espectro_vazio_shifted;
    // Manual fftshift (swap quadrants)
    cv::Mat topLeft = espectro_vazio(cv::Rect(0, 0, N_grid/2, N_grid/2)).clone();
    cv::Mat topRight = espectro_vazio(cv::Rect(N_grid/2, 0, N_grid - N_grid/2,
N_grid/2)).clone();
    cv::Mat bottomLeft = espectro_vazio(cv::Rect(0, N_grid/2, N_grid/2, N_grid -
N_grid/2)).clone();
    cv::Mat bottomRight = espectro_vazio(cv::Rect(N_grid/2, N_grid/2, N_grid - N_grid/2,
N_grid - N_grid/2)).clone();
    cv::Mat shifted(N_grid, N_grid, CV_64FC2);
    bottomRight.copyTo(shifted(cv::Rect(0, 0, N_grid - N_grid/2, N_grid - N_grid/2)));
    bottomLeft.copyTo(shifted(cv::Rect(N_grid - N_grid/2, 0, N_grid/2, N_grid - N_grid/2)));
    topRight.copyTo(shifted(cv::Rect(0, N_grid - N_grid/2, N_grid - N_grid/2, N_grid/2)));
    topLeft.copyTo(shifted(cv::Rect(N_grid - N_grid/2, N_grid - N_grid/2, N_grid/2,
N_grid/2)));

    cv::Mat onda_2d_complex;
    cv::idft(shifted, onda_2d_complex, cv::DFT_SCALE | cv::DFT_COMPLEX_OUTPUT);
    cv::Mat onda_2d_channels[2];
    cv::split(onda_2d_complex, onda_2d_channels);
    cv::Mat onda_2d = onda_2d_channels[0];  // real part

    cv::Mat onda_vis;
    cv::normalize(onda_2d, onda_vis, 0, 255, cv::NORM_MINMAX, CV_8U);
```

```
    // Magnitude of spectrum
    cv::Mat espectro_mag;
    cv::Mat espectro_channels[2];
    cv::split(espectro_vazio, espectro_channels);
    cv::magnitude(espectro_channels[0], espectro_channels[1], espectro_mag);
    cv::Mat espectro_vis;
    cv::normalize(espectro_mag, espectro_vis, 0, 255, cv::NORM_MINMAX, CV_8U);

    // Destaque visual do ponto
    cv::Mat espectro_color;
    cv::cvtColor(espectro_vis, espectro_color, cv::COLOR_GRAY2BGR);
    cv::circle(espectro_color, cv::Point(N_grid/2 - u0, N_grid/2 - v0), 2, cv::Scalar(0, 0,
255), -1);

    std::vector<mm::Image> imgs;
    imgs.push_back(mm::Image(espectro_color));
    imgs.push_back(mm::Image(onda_vis));
    std::vector<std::string> titles;
    titles.push_back("Espectro (1 ponto ativo)");
    titles.push_back("Onda 2D Resultante (IDFT)");
    mm::show(imgs, MM_OUT, titles, 2);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(espectro_color, "tmp/fig_05_grade_2d_0.png");
mm::write(onda_vis, "tmp/fig_05_grade_2d_1.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_05_grade_2d.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_grade_2d.cpp -o
tmp/fig_05_grade_2d -lopencv_stitching -lopencv_alphamat -lopencv_aruco -lopencv_barcode
-lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_grade_2d \
  && test -f "tmp/fig_05_grade_2d.png" \
  || echo "  mm::show não gravou tmp/fig_05_grade_2d.png"
```

```
[1] Espectro (1 ponto ativo)
[2] Onda 2D Resultante (IDFT)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_05_grade_2d_0.png"),
            mm.read("tmp/fig_05_grade_2d_1.png"),
        ],
        titles=[
```

```
            'Espectro (1 ponto ativo)',
            'Onda 2D Resultante (IDFT)',
        ],
        cols=2,
        figsize=(10, 4),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_05_grade_2d_0.png
    (ver a versao Python)")
```

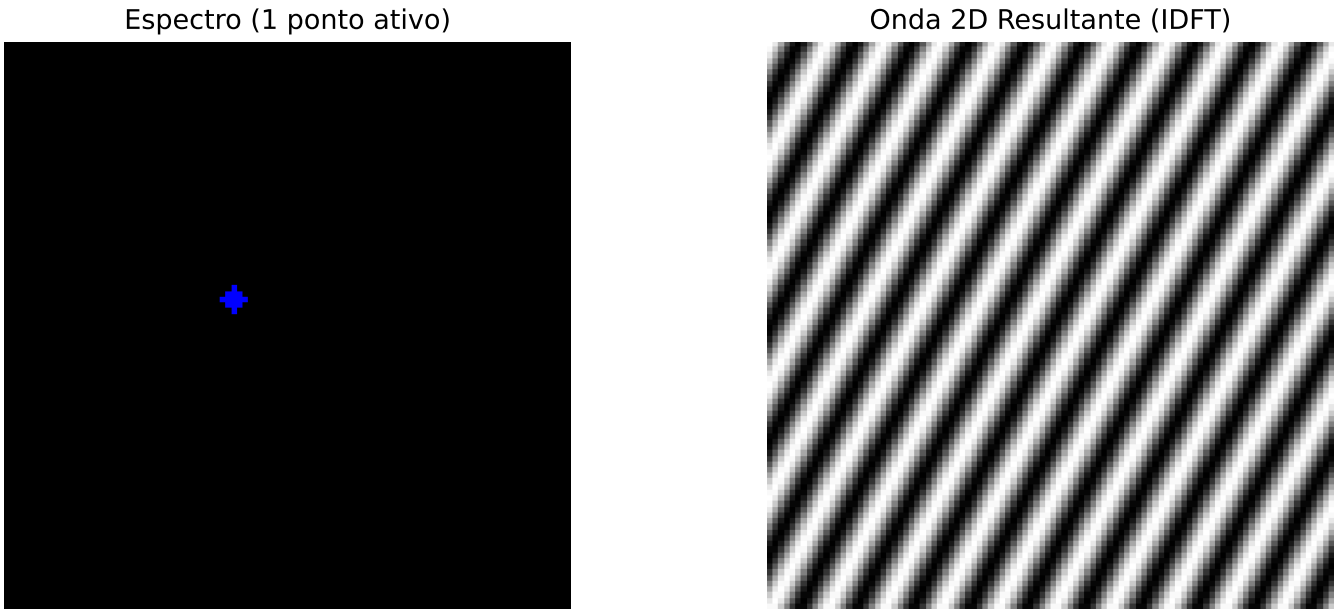


Figure 5.3: Toda frequência no espectro (ponto isolado) corresponde a uma onda senoidal 2D rotacionada no domínio espacial.

5.3.4 Mathematical Definition

Consider an image $f(x, y)$ with dimensions $M \times N$. Its **2D Discrete Fourier Transform** (DFT) is defined by:

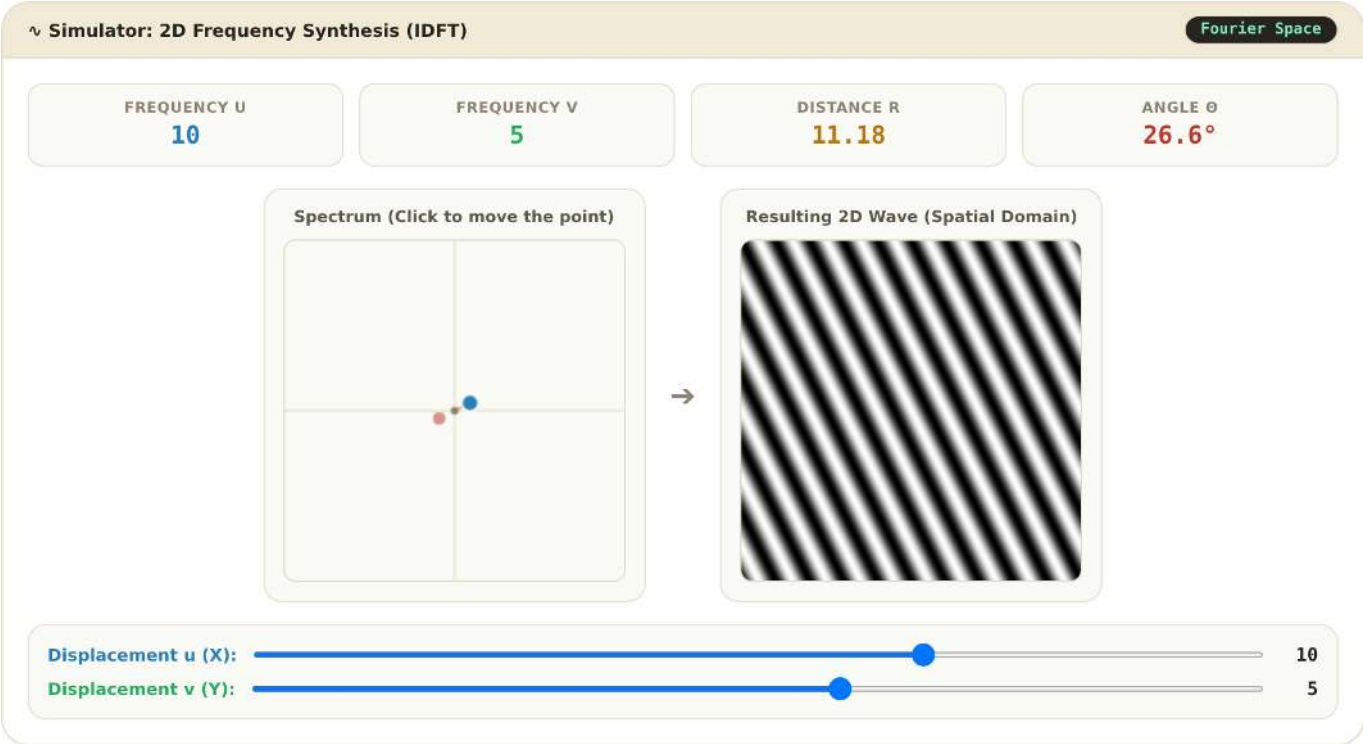
$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y)\, e^{-j2\pi(\frac{ux}{M} + \frac{vy}{N})} \tag{5.1}$$

where $u = 0, 1, \ldots, M-1$ and $v = 0, 1, \ldots, N-1$ represent the discrete frequencies in the horizontal and vertical directions, respectively. The exponential term corresponds to a two-dimensional sinusoid, whose frequency and orientation are determined by the indices (u, v).

The **2D Inverse Discrete Fourier Transform** (IDFT) reconstructs the original image from its coefficients:

$$f(x, y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v)\, e^{j2\pi(\frac{ux}{M} + \frac{vy}{N})} \tag{5.2}$$

Equations Equation 5.1 and Equation 5.2 show that the DFT and the IDFT form a pair of transformations: the former converts the image into the frequency domain, while the latter exactly reconstructs the original image from its coefficients.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap05/sim-05-grade-2d.html

Figure 5.4: Interactive simulator of 2D Fourier synthesis. Change the horizontal (u) and vertical (v) position of the coefficient in the centered frequency spectrum and observe how the distance from the center dictates the spatial frequency (thickness) and the angle dictates the orientation of the generated sine wave.

i On the symbol j

The term j denotes the **imaginary unit**, defined by $j^2 = -1$. In engineering and signal processing, j is adopted instead of i to avoid conflict with the notation for electric current. Its use in the complex exponential, governed by Euler's formula ($e^{j\theta} = \cos\theta + j\sin\theta$), allows for a compact representation of the amplitude and phase of each spatial frequency present in the image.

i What is the DC component?

The coefficient $F(0,0)$, termed the **DC component** (*Direct Current*), equals the sum of the intensities of all pixels in the image (see Figure 5.5):

$$F(0,0) = MN\,\bar{f},$$

where $\bar{f}$ is the average intensity of the image. Therefore, the DC component represents the mean intensity level and, in most natural images, holds the largest magnitude of the spectrum.
The remaining coefficients represent variations around this mean. After applying the **FFT Shift**, the DC component is moved to the center of the spectrum, concentrating low frequencies in the central region and high frequencies at the edges.

5.3.5 Magnitude and Phase

Each coefficient of the Discrete Fourier Transform (DFT) is a complex number and can be written as

$$F(u,v) = R(u,v) + j\,I(u,v),$$

where $R(u,v)$ and $I(u,v)$ correspond, respectively, to the **real** and **imaginary** parts of the coefficient. From

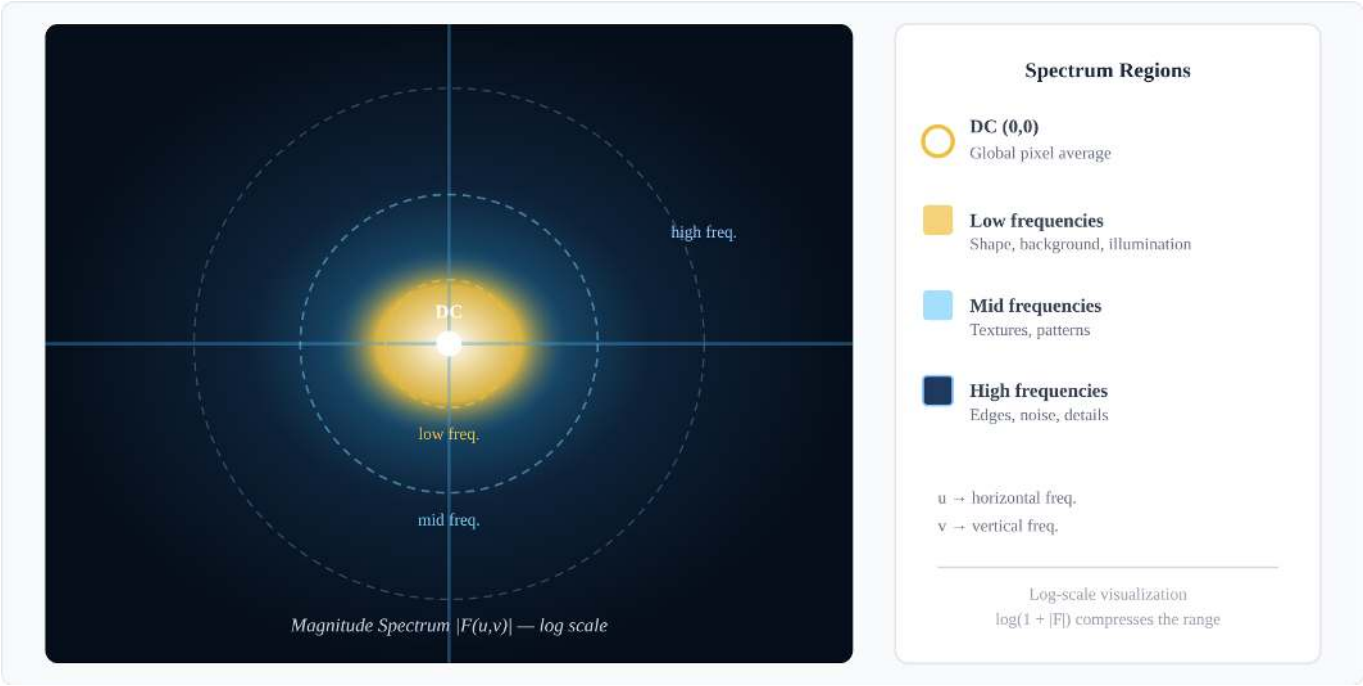


Figure 5.5: Conceptual diagram of the centered 2D Fourier spectrum.

Equation Equation 5.1, we obtain

$$R(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \cos\Big(2\pi \left(\frac{ux}{M} + \frac{vy}{N}\right)\Big),$$

and

$$I(u,v) = -\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \sin\Big(2\pi \left(\frac{ux}{M} + \frac{vy}{N}\right)\Big).$$

From this representation, two fundamental quantities are defined:

- **Magnitude**, which indicates the intensity of the frequency component,

$$|F(u,v)| = \sqrt{R(u,v)^2 + I(u,v)^2};$$

- **Phase**, which determines the spatial alignment (or displacement) of the component,

$$\phi(u,v) = \operatorname{atan2}(I(u,v),\, R(u,v)).$$

Thus, each coefficient can also be written in its polar form,

$$F(u,v) = |F(u,v)|\, e^{j\phi(u,v)}.$$

The Fourier spectrum can, therefore, be visualized by means of two distinct images: the **magnitude spectrum**, typically used to analyze the distribution of frequencies, and the **phase spectrum**, which describes the spatial organization of the sinusoidal components.

Although the magnitude spectrum is the most used for visual inspection, the phase contains a large portion of the structural information of the image. The combination of magnitude and phase allows the exact reconstruction of the original image via the IDFT.

5.3.6 What Do Magnitude and Phase Carry?

A classic demonstration consists of combining the magnitude of one image with the phase of another and reconstructing the result. This experiment highlights that:

- **Phase** preserves the spatial structure of the image, including the position of objects, their contours, and their geometry. Small changes in phase can lead to major visual differences.
- **Magnitude** controls how energy is distributed among spatial frequencies, primarily influencing contrast and texture.

When an image is reconstructed with the magnitude of A and the phase of B, the result tends to **resemble B more than A**, showing that phase is the main component responsible for the spatial organization of the scene. However, magnitude remains important, as it modulates the contrast of the reconstructed structures. Thus, a faithful reconstruction depends on the consistent combination of magnitude and phase.

An example of this behavior is presented in Figure 5.6.

> **i** Analogy with Audio: Limitations and Caveats
>
> The phase of a signal plays distinct roles in audio and images:
> - **Stereo or multichannel audio:** the relative phase between channels is fundamental for perceiving the position of sound sources, through interaural time differences (ITD).
> - **Monaural audio:** absolute phase has little direct perceptual influence.
> - **Images (DFT):** phase is the main factor responsible for the spatial organization of the scene, while magnitude modulates contrast and the distribution of energy across frequencies.
>
> In both domains, **magnitude** is related to the intensity of frequency components: in audio, it influences timbre and perceived loudness; in images, it influences contrast and texture.

```
%%writefile tmp/fig_05_dft_intro.cpp
#define MM_OUT "tmp/fig_05_dft_intro.png"
//| label: fig-05-dft-intro
//| fig-cap: "Phase swap experiment: Image A (coins) and Image B (geometric pattern)
reconstructed with swapped magnitudes and phases. The result shows that the visual structure
is **much more sensitive to phase** than to magnitude: when B's phase is kept, the resulting
image preserves B's spatial organization, even with A's magnitude. Magnitude, in turn, mainly
influences contrast and texture. Note that the reconstruction quality is not perfect - there
are visible artifacts -, evidencing the interdependence between phase and magnitude for a
faithful image representation."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
#include <filesystem>
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <complex>
#include "morph.hpp"

// Helper: manual fftshift (swap quadrants)
void fftshift(cv::Mat& mat) {
    int cx = mat.cols / 2;
    int cy = mat.rows / 2;
    cv::Mat q0(mat, cv::Rect(0, 0, cx, cy));   // Top-Left
    cv::Mat q1(mat, cv::Rect(cx, 0, cx, cy));  // Top-Right
    cv::Mat q2(mat, cv::Rect(0, cy, cx, cy));  // Bottom-Left
    cv::Mat q3(mat, cv::Rect(cx, cy, cx, cy)); // Bottom-Right
    cv::Mat tmp;
```

```
    q0.copyTo(tmp); q3.copyTo(q0); tmp.copyTo(q3);
    q1.copyTo(tmp); q2.copyTo(q1); tmp.copyTo(q2);
}

// Helper: ifftshift (inverse of fftshift)
void ifftshift(cv::Mat& mat) {
    fftshift(mat);  // same operation for even sizes
}

int main() {
    //   Experiment: The Importance of Phase
    //   Image loading
    std::string url     =
"https://upload.wikimedia.org/wikipedia/commons/2/25/GAZI.MD.AHAD_11.jpg";
    std::string caminho = "imagens/coins.jpg";

    cv::Mat img_obj;
    if (!std::filesystem::exists(caminho)) {
        std::filesystem::create_directories("imagens");
        img_obj = mm::read(url);
        mm::write(img_obj, caminho);
    } else {
        img_obj = mm::read(caminho);
    }

    // Convert to grayscale
    cv::Mat img_gray_mat = mm::gray(img_obj);
    mm::Image img_gray = img_gray_mat;

    // Resize to 400x400
    cv::Mat img_a_mat;
    cv::resize(img_gray_mat, img_a_mat, cv::Size(400, 400));
    mm::Image img_a = img_a_mat;

    // Create synthetic image B (geometric pattern)
    cv::Mat img_b_mat = cv::Mat::zeros(400, 400, CV_8UC1);
    cv::rectangle(img_b_mat, cv::Point(100, 100), cv::Point(300, 300), 255, -1);
    cv::circle(img_b_mat, cv::Point(200, 200), 150, 128, 10);
    mm::Image img_b = img_b_mat;

    // Compute FFTs
    cv::Mat img_a_32f, img_b_32f;
    img_a_mat.convertTo(img_a_32f, CV_32F);
    img_b_mat.convertTo(img_b_32f, CV_32F);

    cv::Mat FA_complex, FB_complex;
    cv::dft(img_a_32f, FA_complex, cv::DFT_COMPLEX_OUTPUT);
    cv::dft(img_b_32f, FB_complex, cv::DFT_COMPLEX_OUTPUT);

    // Split into channels for magnitude/phase manipulation
    std::vector<cv::Mat> FA_channels(2), FB_channels(2);
    cv::split(FA_complex, FA_channels);
    cv::split(FB_complex, FB_channels);

    // Compute magnitude and phase
    cv::Mat FA_mag, FA_phase, FB_mag, FB_phase;
    cv::magnitude(FA_channels[0], FA_channels[1], FA_mag);
    cv::phase(FA_channels[0], FA_channels[1], FA_phase);
    cv::magnitude(FB_channels[0], FB_channels[1], FB_mag);
    cv::phase(FB_channels[0], FB_channels[1], FB_phase);
```

```
    // Reconstruct with swapped magnitudes and phases
    cv::Mat rec_A_mag_B_fase_complex_real, rec_A_mag_B_fase_complex_imag;
    cv::Mat rec_B_mag_A_fase_complex_real, rec_B_mag_A_fase_complex_imag;

    // rec_A_mag_B_fase = |FA| * exp(i * angle(FB))
    cv::Mat real_A = FA_mag.mul(cv::Mat(FA_phase.size(), CV_32F, cv::Scalar(0)));
    cv::Mat real_B = FB_mag.mul(cv::Mat(FB_phase.size(), CV_32F, cv::Scalar(0)));
    // For complex multiplication: (r1*c1 - i1*s1, r1*s1 + i1*c1)
    cv::Mat r1 = FA_mag, i1 = cv::Mat::zeros(FA_mag.size(), CV_32F);
    cv::Mat c1, s1;
    cv::phase(FB_channels[0], FB_channels[1], FB_phase, true);
    cv::Mat ones = cv::Mat::ones(FB_phase.size(), CV_32F);
    cv::Mat cos_phase, sin_phase;
    cv::Mat fb_phase_deg = FB_phase * (CV_PI / 180.0);
    cv::Mat fa_phase_deg = FA_phase * (CV_PI / 180.0);
    cv::polarToCart(ones, fb_phase_deg, c1, s1);

    // Complex multiply: (r1 + j*i1) * (c1 + j*s1) = (r1*c1 - i1*s1) + j*(r1*s1 + i1*c1)
    cv::Mat resp_real = r1.mul(c1) - i1.mul(s1);
    cv::Mat resp_imag = r1.mul(s1) + i1.mul(c1);

    // For rec_B_mag_A_fase: |FB| * exp(i * angle(FA))
    cv::Mat r2 = FB_mag, i2 = cv::Mat::zeros(FB_mag.size(), CV_32F);
    cv::Mat c2, s2;
    cv::polarToCart(cv::Mat::ones(fa_phase_deg.size(), CV_32F), fa_phase_deg, c2, s2);
    cv::Mat resp2_real = r2.mul(c2) - i2.mul(s2);
    cv::Mat resp2_imag = r2.mul(s2) + i2.mul(c2);

    // Combine real and imaginary into complex matrices
    cv::Mat rec_A_mag_B_fase_complex, rec_B_mag_A_fase_complex;
    std::vector<cv::Mat> rec_A_channels = {resp_real, resp_imag};
    std::vector<cv::Mat> rec_B_channels = {resp2_real, resp2_imag};
    cv::merge(rec_A_channels, rec_A_mag_B_fase_complex);
    cv::merge(rec_B_channels, rec_B_mag_A_fase_complex);

    // Inverse FFT
    cv::Mat rec_A_mag_B_fase_float, rec_B_mag_A_fase_float;
    cv::idft(rec_A_mag_B_fase_complex, rec_A_mag_B_fase_float, cv::DFT_SCALE |
cv::DFT_REAL_OUTPUT);
    cv::idft(rec_B_mag_A_fase_complex, rec_B_mag_A_fase_float, cv::DFT_SCALE |
cv::DFT_REAL_OUTPUT);

    // Convert to 8-bit
    cv::Mat rec_A_mag_B_fase_mat, rec_B_mag_A_fase_mat;
    rec_A_mag_B_fase_float.convertTo(rec_A_mag_B_fase_mat, CV_8U);
    rec_B_mag_A_fase_float.convertTo(rec_B_mag_A_fase_mat, CV_8U);

    mm::Image rec_A_mag_B_fase = rec_A_mag_B_fase_mat;
    mm::Image rec_B_mag_A_fase = rec_B_mag_A_fase_mat;

    // Display results
    mm::show(
        {img_a, img_b, rec_A_mag_B_fase, rec_B_mag_A_fase},
        MM_OUT,
        {"Imagem A", "Imagem B", "Mag(A) + Fase(B)", "Mag(B) + Fase(A)"},
        4
    );

    std::cout << "  A fase preserva bordas e contornos; a magnitude controla contraste e" <<
std::endl;
```

```
    std::cout << "textura. Em áudio estéreo, a fase afeta a localização espacial; em" <<
std::endl;
    std::cout << "imagens, determina a organização da cena." << std::endl;


// [pdi:state-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp/state");
mm::write(img_gray, "tmp/state/img_gray_20.png");
// [pdi:state-io:end]

// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_a, "tmp/fig_05_dft_intro_0.png");
mm::write(img_b, "tmp/fig_05_dft_intro_1.png");
mm::write(rec_A_mag_B_fase, "tmp/fig_05_dft_intro_2.png");
mm::write(rec_B_mag_A_fase, "tmp/fig_05_dft_intro_3.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_05_dft_intro.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_dft_intro.cpp -o
tmp/fig_05_dft_intro -lopencv_stitching -lopencv_alphamat -lopencv_aruco -lopencv_barcode
-lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_dft_intro \
  && test -f "tmp/fig_05_dft_intro.png" \
  || echo "  mm::show não gravou tmp/fig_05_dft_intro.png"
```

```
[1] Imagem A
[2] Imagem B
[3] Mag(A) + Fase(B)
[4] Mag(B) + Fase(A)
 A fase preserva bordas e contornos; a magnitude controla contraste e
textura. Em áudio estéreo, a fase afeta a localização espacial; em
imagens, determina a organização da cena.
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_05_dft_intro_0.png"),
            mm.read("tmp/fig_05_dft_intro_1.png"),
            mm.read("tmp/fig_05_dft_intro_2.png"),
            mm.read("tmp/fig_05_dft_intro_3.png"),
        ],
        titles=[
            'Imagem A',
            'Imagem B',
            'Mag(A) + Fase(B)',
            'Mag(B) + Fase(A)',
```

```
        ],
        cols=4,
        figsize=(16, 4),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_05_dft_intro_0.png
    (ver a versao Python)")
```

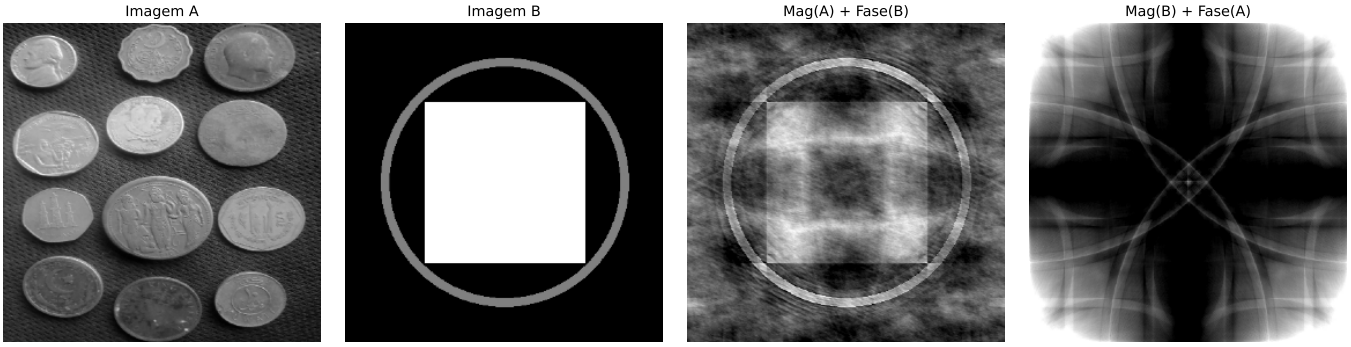


Figure 5.6: Experimento de troca de fase: Imagem A (moedas) e Imagem B (padrão geométrico) reconstruídas com magnitudes e fases trocadas. O resultado mostra que a estrutura visual é **muito mais sensível à fase** do que à magnitude: quando a fase de B é mantida, a imagem resultante preserva a organização espacial de B, mesmo com a magnitude de A. A magnitude, por sua vez, influencia principalmente o contraste e a textura. Observe que a qualidade da reconstrução não é perfeita — há artefatos visíveis —, evidenciando a interdependência entre fase e magnitude para uma representação fiel da imagem.

5.4 Convolution Theorem and Filtering Strategies

The **Convolution Theorem** establishes a fundamental relationship between the spatial and frequency domains:

$$f(x,y) \circledast h(x,y) \overset{\mathcal{F}}{\longleftrightarrow} F(u,v)\,H(u,v) \tag{5.3}$$

where $\circledast$ denotes **discrete circular convolution**. Thus, the convolution between an image $f(x,y)$ and a filter $h(x,y)$ can be replaced by the multiplication of their spectra.

In practice, to obtain the same result as the linear convolution performed in the spatial domain, **zero-padding** is applied before the Fast Fourier Transform (FFT), avoiding artifacts at the image borders.

However, frequency-domain filtering is not always the most efficient alternative. For filters such as the Gaussian and the Box Filter, the **separability** property allows a significant reduction in the computational cost of convolution in the spatial domain.

5.4.1 Separable vs. Non-Separable *Kernel*

A ***separable kernel*** can be written as the outer product of two one-dimensional vectors,

$$H = v\,h^T,$$

allowing the two-dimensional convolution to be replaced by two consecutive one-dimensional convolutions: one in the horizontal direction and another in the vertical direction.

In contrast, a ***non-separable kernel*** does not admit such a decomposition, and therefore its convolution must be performed directly over the two-dimensional neighborhood.

In practice, for a kernel of dimension $K \times K$, direct convolution requires K^2 multiplications per pixel, whereas a separable kernel requires only $2K$ multiplications, significantly reducing the computational cost.

5.4.2 Computational Efficiency Analysis

Consider an image of dimensions $M \times N$ and a square filter of size $K \times K$. Table 5.2 compares the complexity of the main filtering strategies.

Table 5.2: Comparison of the complexity of direct convolution, separable convolution, and filtering via the Fast Fourier Transform (FFT).

Filtering Method	Asymptotic Complexity	Dependence on K	Typical Application
Non-separable spatial	$\mathcal{O}(MNK^2)$	Quadratic	Small, non-separable *kernels*
Separable spatial	$\mathcal{O}(MNK)$	Linear	Gaussian and mean filters
Via FFT	$\mathcal{O}(MN\log(MN))$	Independent of K	Large *kernels*

For small *kernels*, spatial convolution, especially when the filter is separable, tends to be more efficient due to the low cost of operations. As the *kernel* size increases, FFT-based filtering becomes more advantageous, since its cost is practically independent of the filter dimension.

5.4.3 Discussion of experimental results

The graph obtained in the test with the coin image (2560×1920), presented in Figure 5.7, confirms the behavior predicted by the computational complexity analysis.

1. **Non-separable convolution ($\mathcal{O}(MNK^2)$)**
   Direct convolution exhibits quadratic growth with kernel size. For small values of K, the cost is low, but it increases rapidly as the kernel grows, becoming unfeasible for real-time applications.
2. **FFT-based filtering ($\mathcal{O}(MN\log(MN))$)**
   The cost of the FFT depends only on the image size, being independent of K. Therefore, its performance remains approximately constant as the kernel varies, making it advantageous for large or non-separable filters.
3. **Separable convolution ($\mathcal{O}(MNK)$)**
   Decomposing the kernel into two one-dimensional filters significantly reduces the computational cost. In practice, this approach tends to be the most efficient for separable filters, especially in optimized implementations.

In general, the choice of method depends on the size and structure of the kernel. Separable filters are more efficient in the spatial domain, while the FFT becomes more advantageous for large kernels or multiple convolutions in the frequency domain.

$$g = \mathcal{F}^{-1}[\mathcal{F}(f) \cdot \mathcal{F}(h)] \quad \text{(FFT)} \qquad g = f \circledast h \quad \text{(direct convolution)} \qquad g = (f \circledast v) \circledast h^T \quad \text{(separable)} \tag{5.4}$$

where:

- $f(x, y)$ represents the input image;
- $h(x, y)$ is the two-dimensional filter kernel;
- v and h^T are, respectively, the vertical and horizontal vectors that compose the separable kernel.

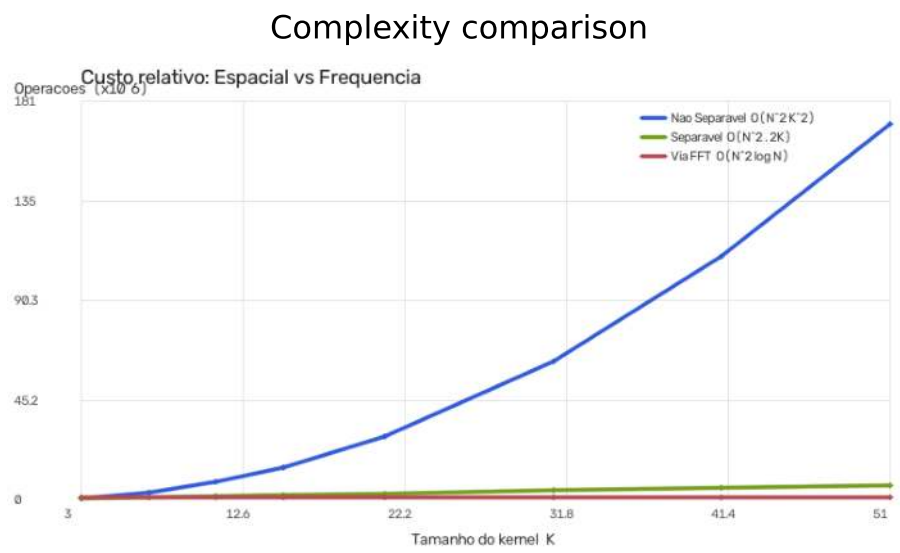


Figure 5.7: Efficiency comparison: Non-separable Convolution (Spatial 2D), Separable (Spatial 1D), and via FFT.

! The Problem of Circular Convolution (*Wrap-Around*)

The Discrete Fourier Transform (DFT) assumes that the image is **periodically extended in space**, meaning that its borders repeat indefinitely.
Under this condition, multiplication in the frequency domain corresponds to a **circular convolution** in the spatial domain. Consequently, opposite regions of the image (top and bottom, left and right) begin to interact artificially, as illustrated in Figure 5.8.
Applying zero-padding before the FFT reduces this effect by extending the image with null values at the borders, approximating the result of linear convolution. This behavior can be interpreted in light of the Convolution Theorem, presented in Figure 5.9.

```
%%writefile tmp/fig_05_padding_error.cpp
#define MM_OUT "tmp/fig_05_padding_error.png"
#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include "morph.hpp"
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_20.png");
// [pdi:state-io:end]

    //| label: fig-05-padding-error
    //| fig-cap: "Without *padding*, a severe shift causes the image to leak to the opposite
    side (circular convolution)."
    //| echo: true
    //| output: true

    //   Define M and N
    int M = img_gray.h;
    int N = img_gray.w;

    // Convert to CV_64F for complex operations
```

```
    cv::Mat img_gray_cv;
    cv::Mat img_gray_disp = cv::Mat(img_gray.h, img_gray.w, CV_8UC1, img_gray.data.data());
    img_gray_disp.convertTo(img_gray_cv, CV_64F);

    // Simulation of a brutal shift filter
    cv::Mat H_shift(M, N, CV_64FC2, cv::Scalar(0, 0));
    for (int u = 0; u < M; u++) {
        for (int v = 0; v < N; v++) {
            double angle = -2.0 * CV_PI * (u * 120.0 / M + v * 120.0 / N);
            H_shift.at<cv::Vec2d>(u, v)[0] = cos(angle);  // Real part
            H_shift.at<cv::Vec2d>(u, v)[1] = sin(angle);  // Imaginary part
        }
    }

    // Filtering WITHOUT padding (causes wrap-around)
    cv::Mat img_gray_32f;
    img_gray_disp.convertTo(img_gray_32f, CV_32F);
    cv::Mat F_img, F_padded;

    // Compute FFT of the image
    cv::Mat planes[] = {img_gray_32f, cv::Mat::zeros(img_gray_32f.size(), CV_32F)};
    cv::Mat img_complex;
    cv::merge(planes, 2, img_complex);
    cv::dft(img_complex, F_img, cv::DFT_COMPLEX_OUTPUT);

    // Perform pointwise multiplication in frequency domain
    cv::Mat H_shift_32f, F_img_shifted;
    H_shift.convertTo(H_shift_32f, CV_32FC2);
    cv::mulSpectrums(F_img, H_shift_32f, F_img_shifted, 0);

    // Inverse FFT to get the filtered image
    cv::Mat img_vazada;
    cv::idft(F_img_shifted, img_vazada, cv::DFT_SCALE | cv::DFT_REAL_OUTPUT);

    // Collect real part (already done by DFT_REAL_OUTPUT)
    cv::Mat img_vazada_vis;
    cv::normalize(img_vazada, img_vazada_vis, 0, 255, cv::NORM_MINMAX, CV_8U);

    // Display results
    cv::Mat display_original = cv::Mat(img_gray.h, img_gray.w, CV_8UC1, img_gray.data.data());

    std::vector<mm::Image> images;
    images.push_back(mm::Image(display_original));
    images.push_back(mm::Image(img_vazada_vis));

    std::vector<std::string> titles;
    titles.push_back("Original");
    titles.push_back("Filtering w/o Padding (Leakage)");

    mm::show(images, MM_OUT, titles, 2);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray, "tmp/fig_05_padding_error_0.png");
mm::write(img_vazada_vis, "tmp/fig_05_padding_error_1.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_05_padding_error.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_padding_error.cpp -o
tmp/fig_05_padding_error -lopencv_stitching -lopencv_alphamat -lopencv_aruco -lopencv_barcode
-lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_padding_error \
  && test -f "tmp/fig_05_padding_error.png" \
  || echo "  mm::show não gravou tmp/fig_05_padding_error.png"
```

```
[1] Original
[2] Filtering w/o Padding (Leakage)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_05_padding_error_0.png"),
            mm.read("tmp/fig_05_padding_error_1.png"),
        ],
        titles=[
            'Original',
            'Filtragem s/ Padding (Vazamento)',
        ],
        cols=2,
        figsize=(10, 4),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_padding_error_0.png (ver a versao Python)")
```

Figure 5.8: Sem *padding*, um deslocamento severo faz a imagem vazar para o lado oposto (convolução circular).

```
%%writefile tmp/fig_05_conv_teorema.cpp
#define MM_OUT "tmp/fig_05_conv_teorema.png"
//| label: fig-05-conv-teorema
//| fig-cap: "Teorema da Convolução: filtrar no domínio do espaço (Gaussiana) equivale a
multiplicar o espectro por $H(u,v)$ na frequência. As saídas coincidem visualmente."
//| echo: true
//| output: true

// Same smoothing through two paths: space vs. frequency.
#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include "morph.hpp"
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_20.png");
// [pdi:state-io:end]

    // img_gray is injected here (mm::Image)

    int M = img_gray.h;  // height
    int N = img_gray.w;  // width

    cv::Mat H = mm::gaussFilter(M, N, 30);      // H(u,v): gaussian low-pass transfer function
    mm::Image f_freq = mm::freqFilter(img_gray, H);    // convolution via frequency-domain
multiplication

    // Same smoothing done in the spatial domain: Gaussian blur with kernel 31x31, sigma=5
    cv::Mat img_mat = img_gray;  // convert mm::Image to cv::Mat
    cv::Mat blurred;
    cv::GaussianBlur(img_mat, blurred, cv::Size(31, 31), 5);
```

```
    mm::Image f_esp(blurred);

    mm::show(
        std::vector<mm::Image>{img_gray, f_esp, f_freq},
        MM_OUT,
        std::vector<std::string>{"Original", "Espaco: Gaussiana", "Frequencia:
H(u,v).F(u,v)"},
        3
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray, "tmp/fig_05_conv_teorema_0.png");
mm::write(f_esp, "tmp/fig_05_conv_teorema_1.png");
mm::write(f_freq, "tmp/fig_05_conv_teorema_2.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_05_conv_teorema.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_conv_teorema.cpp -o
tmp/fig_05_conv_teorema -lopencv_stitching -lopencv_alphamat -lopencv_aruco -lopencv_barcode
-lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_conv_teorema \
  && test -f "tmp/fig_05_conv_teorema.png" \
  || echo "  mm::show não gravou tmp/fig_05_conv_teorema.png"
```

```
[1] Original
[2] Espaco: Gaussiana
[3] Frequencia: H(u,v).F(u,v)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_05_conv_teorema_0.png"),
            mm.read("tmp/fig_05_conv_teorema_1.png"),
            mm.read("tmp/fig_05_conv_teorema_2.png"),
        ],
        titles=[
            'Original',
            'Espaco: Gaussiana',
            'Frequencia: H(u,v).F(u,v)',
        ],
        cols=3,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_conv_teorema_0.png (ver a versao Python)")
```

Figure 5.9: Teorema da Convolução: filtrar no domínio do espaço (Gaussiana) equivale a multiplicar o espectro por $H(u, v)$ na frequência. As saídas coincidem visualmente.

> **i** On Numerical Difference
>
> The residual difference on the order of 10^{-13} does not violate the Convolution Theorem, but rather reflects **computational limitations** inherent to floating-point arithmetic (double precision, ~10^{-16}) and to the **order of operations** between the two methods:
>
> - **Spatial convolution:** weighted sum of neighbors with successive roundings.
> - **Frequency convolution:** involves three FFT transforms and one complex multiplication, subject to truncation and quantization errors.
>
> Therefore, the theoretical equality is exact, but the numerical implementation produces a practically negligible difference (relative error $< 10^{-12}$), confirming the theorem within machine precision.

5.5 Frequency Domain Filters

A frequency domain filter can be interpreted as a **transfer function applied to the image spectrum**. In this representation, each frequency coefficient is multiplied by a value between 0 and 1, which determines its attenuation or preservation. The shape of this function defines the visual effect of the filter.

Abrupt cutoff and *ringing*. Ideal filters with an instantaneous transition at a cutoff frequency D_0 produce discontinuities in the frequency domain. This discontinuity is reflected in the spatial domain as oscillations near edges, known as *ringing*. This effect is associated with convolution with functions of infinite support in space, such as the *sinc* function, as illustrated in Figure 5.10.

Filters with smooth transition. Alternatives such as the Gaussian and Butterworth filters smooth the transition between pass and reject regions, reducing *ringing*. In contrast, this smoothing implies a less defined separation boundary between preserved and attenuated frequencies.

```
%%writefile tmp/fig_05_conv_teorema_zoom.cpp
#define MM_OUT "tmp/fig_05_conv_teorema_zoom.png"
//| label: fig-05-conv-teorema-zoom
//| fig-cap: "A Dualidade Perigosa: o corte abrupto na Frequência (cilindro Ideal) vira
obrigatoriamente uma *sinc* no espaço. Suas ondulações causam o *ringing* fantasma nas bordas
da imagem."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
```

```
#include <vector>
#include <string>
#include "morph.hpp"
#include <filesystem>

int main() {
    // Filtro Ideal na frequência (cilindro) e sua resposta espacial (sinc 2D).
    int N = 128;
    cv::Mat H_freq  = mm::idealFilter(N, N, 20);      // 1 dentro do raio 20, 0 fora
    mm::Image h_space = mm::spatialKernel(H_freq);    // ifft2(H) -> ondulações da sinc
(ringing)

    mm::show(
        std::vector<mm::Image>{H_freq, h_space},
        MM_OUT,
        {
            "Frequencia: filtro Ideal (cilindro)",
            "Espaco: ondulacoes da sinc (causa do ringing)",
        },
        2
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(H_freq, "tmp/fig_05_conv_teorema_zoom_0.png");
mm::write(h_space, "tmp/fig_05_conv_teorema_zoom_1.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_05_conv_teorema_zoom.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_conv_teorema_zoom.cpp -o
tmp/fig_05_conv_teorema_zoom -lopencv_stitching -lopencv_alphamat -lopencv_aruco
-lopencv_barcode -lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_conv_teorema_zoom \
  && test -f "tmp/fig_05_conv_teorema_zoom.png" \
  || echo "  mm::show não gravou tmp/fig_05_conv_teorema_zoom.png"
```

```
[1] Frequencia: filtro Ideal (cilindro)
[2] Espaco: ondulacoes da sinc (causa do ringing)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_05_conv_teorema_zoom_0.png"),
            mm.read("tmp/fig_05_conv_teorema_zoom_1.png"),
        ],
        titles=[
```

```
            'Frequencia: filtro Ideal (cilindro)',
            'Espaco: ondulacoes da sinc (causa do ringing)',
        ],
        cols=2,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_conv_teorema_zoom_0.png (ver a versao Python)")
```

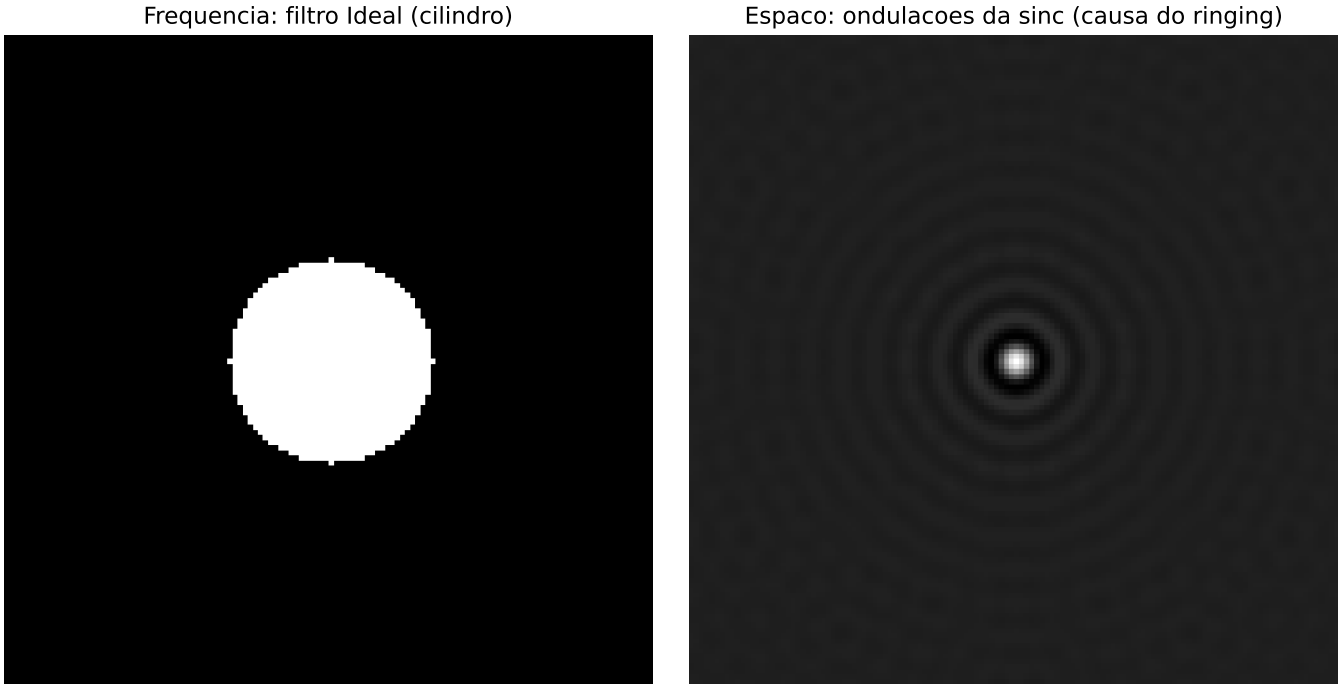


Figure 5.10: A Dualidade Perigosa: o corte abrupto na Frequência (cilindro Ideal) vira obrigatoriamente uma *sinc* no espaço. Suas ondulações causam o *ringing* fantasma nas bordas da imagem.

5.5.1 Low-Pass Filters

Low-pass filters attenuate high-frequency components, resulting in image smoothing and noise reduction. After spectrum centering (FFT Shift), the distance from each point to the center is given by:

$$D(u,v) = \sqrt{\left(u - \tfrac{M}{2}\right)^2 + \left(v - \tfrac{N}{2}\right)^2} \tag{5.5}$$

Ideal Filter (LPFI):

$$H_{\text{ideal}}(u,v) = \begin{cases} 1, & D(u,v) \leq D_0 \\ 0, & D(u,v) > D_0 \end{cases} \tag{5.6}$$

The abrupt cutoff at D_0 introduces discontinuities in the frequency domain, resulting in oscillations in the spatial domain known as *ringing*. This effect is associated with convolution with functions of infinite support.

Gaussian Filter (LPFG):

$$H_{\text{gauss}}(u,v) = e^{-D^2(u,v)/(2\sigma^2)} \tag{5.7}$$

The smoothness of the Gaussian function in the frequency domain avoids discontinuities, which eliminates *ringing* and produces a gradual transition between preserved and attenuated frequencies.

Butterworth Filter (LPFB) of order n:

$$H_{\mathrm{BW}}(u,v) = \frac{1}{1 + [D(u,v)/D_0]^{2n}} \tag{5.8}$$

The parameter n controls the smoothness of the transition between frequency pass and rejection. Small values produce smooth transitions, while large values approximate the behavior of the ideal filter, with a higher risk of *ringing*. A comparative example is presented in Figure 5.11.

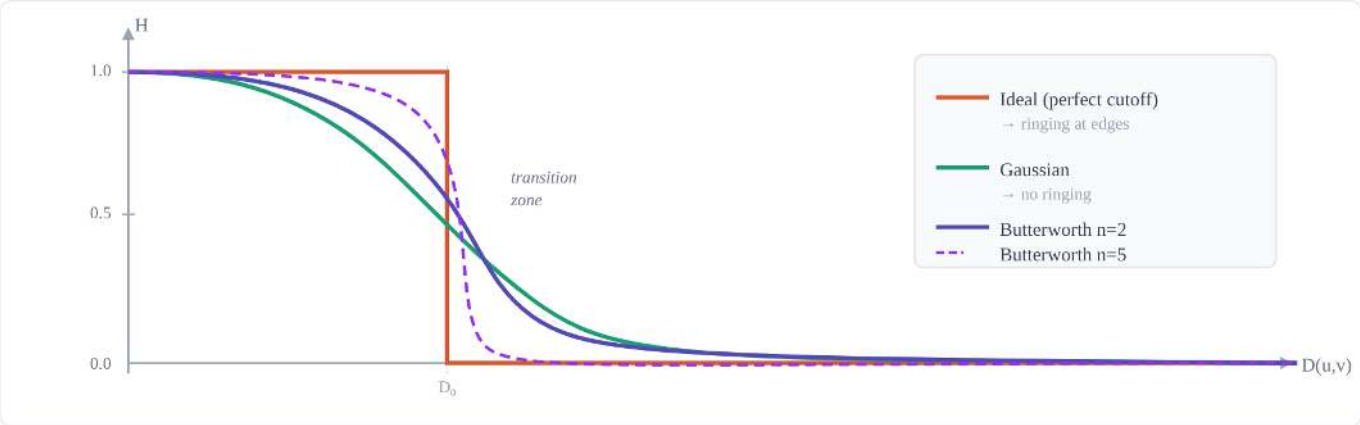


Figure 5.11: Low-pass filters.

5.5.2 High-Pass and Band-Pass Filters

High-pass filters can be obtained from a complementary low-pass filter, defined as:

$$H_{\mathrm{HP}}(u,v) = 1 - H_{\mathrm{LP}}(u,v)$$

This type of filter preserves high-frequency components, enhancing edges and details, while attenuating regions of smooth variation.

Band-pass filters preserve only an intermediate range of frequencies, bounded by two radii D_L and D_H:

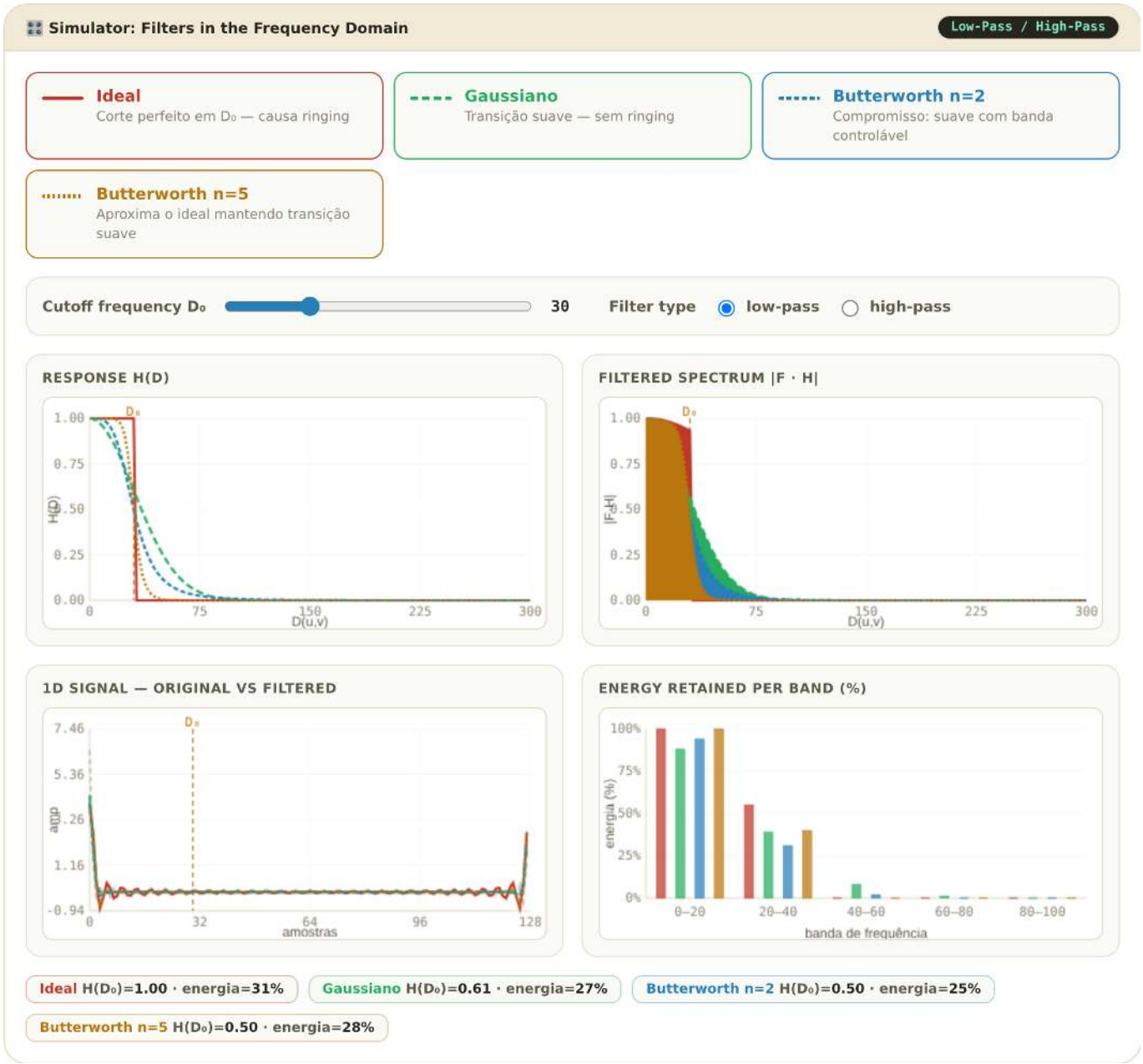
$$H_{\mathrm{BP}}(u,v) = H_{\mathrm{LP}}^{(D_H)}(u,v) \cdot \left[1 - H_{\mathrm{LP}}^{(D_L)}(u,v)\right]$$

This type of filtering is useful when one wishes to simultaneously remove low- and high-frequency components, preserving only structures of intermediate scale.

An important application is the removal of **periodic noise**, in which regular patterns appear as localized peaks in the magnitude spectrum. These peaks can be attenuated by means of *notch* (band-reject) filters, positioned specifically at the undesired frequencies.

Examples of filters in the frequency domain are presented in the simulator of Figure 5.12, Figure 5.13, and Figure 5.14.

```
%%writefile tmp/fig_05_filtros_freq.cpp
#define MM_OUT "tmp/fig_05_filtros_freq.png"
//| label: fig-05-filtros-freq
//| fig-cap: "Comparação entre filtros passa-baixa: Ideal (D =30), Gaussiano (D =30) e
Butterworth (D =30, n=2). Perfis de H(u,v) ao longo de uma linha central e imagens filtradas
correspondentes."
//| echo: true
//| output: true
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap05/sim-05-filtros.html

Figure 5.12: Interactive simulator of filters in the frequency domain.

```cpp
#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <cmath>
#include "morph.hpp"

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_20.png");
// [pdi:state-io:end]

    // img_gray is provided as a valid mm::Image

    int M = img_gray.h;
    int N = img_gray.w;
    int D0 = 30;
    int n_bw = 2;

    //   Transfer functions (centered H) via morph.hpp constructors
    //   Ideal:       1 if D<=D0, else 0
    //   Gaussian:    exp(-D^2/(2 D0^2))
    //   Butterworth: 1 / (1 + (D/D0)^(2n))
    cv::Mat H_ideal = mm::idealFilter(M, N, D0);
    cv::Mat H_gauss = mm::gaussFilter(M, N, D0);
    cv::Mat H_bw    = mm::butterFilter(M, N, D0, n_bw);

    //   Filtered images via FFT
    mm::Image img_ideal = mm::freqFilter(img_gray, H_ideal);
    mm::Image img_gauss = mm::freqFilter(img_gray, H_gauss);
    mm::Image img_bw    = mm::freqFilter(img_gray, H_bw);

    // Helper for visualizing H
    auto H_vis = [](const cv::Mat& H) -> cv::Mat {
        cv::Mat H_scaled = H * 255.0;
        cv::Mat H_uint8;
        H_scaled.convertTo(H_uint8, CV_8U);
        cv::Mat result;
        cv::normalize(H_uint8, result, 0, 255, cv::NORM_MINMAX);
        return result;
    };

    //   H(u,v) profiles on central line, drawn with cv2.line
    int linha = M / 2;
    int Wp = 512, Hp = 288;
    cv::Mat perfil(Hp, Wp, CV_8UC3, cv::Scalar(255, 255, 255));

    auto traca = [&](const cv::Mat& row_ref, const cv::Scalar& cor) {
        // Get the central line values from H
        std::vector<double> row;
        for (int j = 0; j < N; ++j) {
            row.push_back(H_ideal.at<double>(linha, j));  // Note: using H_ideal as reference
for indices
        }
        // Use actual H values passed
        cv::Point ant(-1, -1);
        for (int x = 0; x < N; ++x) {
            int px = (int)std::round(x * (Wp - 1) / (N - 1));
            double val = row_ref.at<double>(0, x);  // Access directly
            if (x == 0) {
                // Need to handle differently - row_ref is actually a full row
                val = row_ref.at<double>(linha, x);
```

```
            }
            int py = (int)std::round(10 + (1.0 - val) * (Hp - 20));
            if (ant.x >= 0) {
                cv::line(perfil, ant, cv::Point(px, py), cor, 2, cv::LINE_AA);
            }
            ant = cv::Point(px, py);
        }
    };

    // Redo traca properly with direct row access
    auto traca2 = [&](const cv::Mat& H, const cv::Scalar& cor) {
        cv::Point ant(-1, -1);
        for (int x = 0; x < N; ++x) {
            int px = (int)std::round(x * (Wp - 1) / (N - 1));
            double val = H.at<double>(linha, x);
            int py = (int)std::round(10 + (1.0 - val) * (Hp - 20));
            if (ant.x >= 0) {
                cv::line(perfil, ant, cv::Point(px, py), cor, 2, cv::LINE_AA);
            }
            ant = cv::Point(px, py);
        }
    };

    traca2(H_ideal, cv::Scalar(216, 90, 48));  // BGR: blue-ish
    traca2(H_gauss, cv::Scalar(29, 158, 117)); // BGR: green-ish
    traca2(H_bw,    cv::Scalar(83, 74, 183));  // BGR: red-ish

    // Convert mm::Image to cv::Mat for display if needed
    cv::Mat img_gray_mat = img_gray;
    cv::Mat img_ideal_mat = img_ideal;
    cv::Mat img_gauss_mat = img_gauss;
    cv::Mat img_bw_mat = img_bw;

    mm::show(
        std::vector<mm::Image>{img_gray, img_ideal, img_gauss, img_bw,
             H_vis(H_ideal), H_vis(H_gauss), H_vis(H_bw), perfil},
        MM_OUT,
        std::vector<std::string>{
            "Original", "LPF Ideal", "LPF Gaussiano", "LPF Butterworth (n=2)",
            "H Ideal", "H Gaussiano", "H Butterworth", "Perfis H(u,v)",
        },
        4
    );

    return 0;
}
```

```
Overwriting tmp/fig_05_filtros_freq.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_filtros_freq.cpp -o
tmp/fig_05_filtros_freq -lopencv_stitching -lopencv_alphamat -lopencv_aruco -lopencv_barcode
-lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_filtros_freq \
  && test -f "tmp/fig_05_filtros_freq.png" \
  || echo "  mm::show não gravou tmp/fig_05_filtros_freq.png"
```

```
[1] Original
[2] LPF Ideal
[3] LPF Gaussiano
[4] LPF Butterworth (n=2)
[5] H Ideal
[6] H Gaussiano
[7] H Butterworth
[8] Perfis H(u,v)
```

```
try:
    mm.show(mm.read("tmp/fig_05_filtros_freq.png"), figsize=(16, 9))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_filtros_freq.png (ver a versao Python)")
```

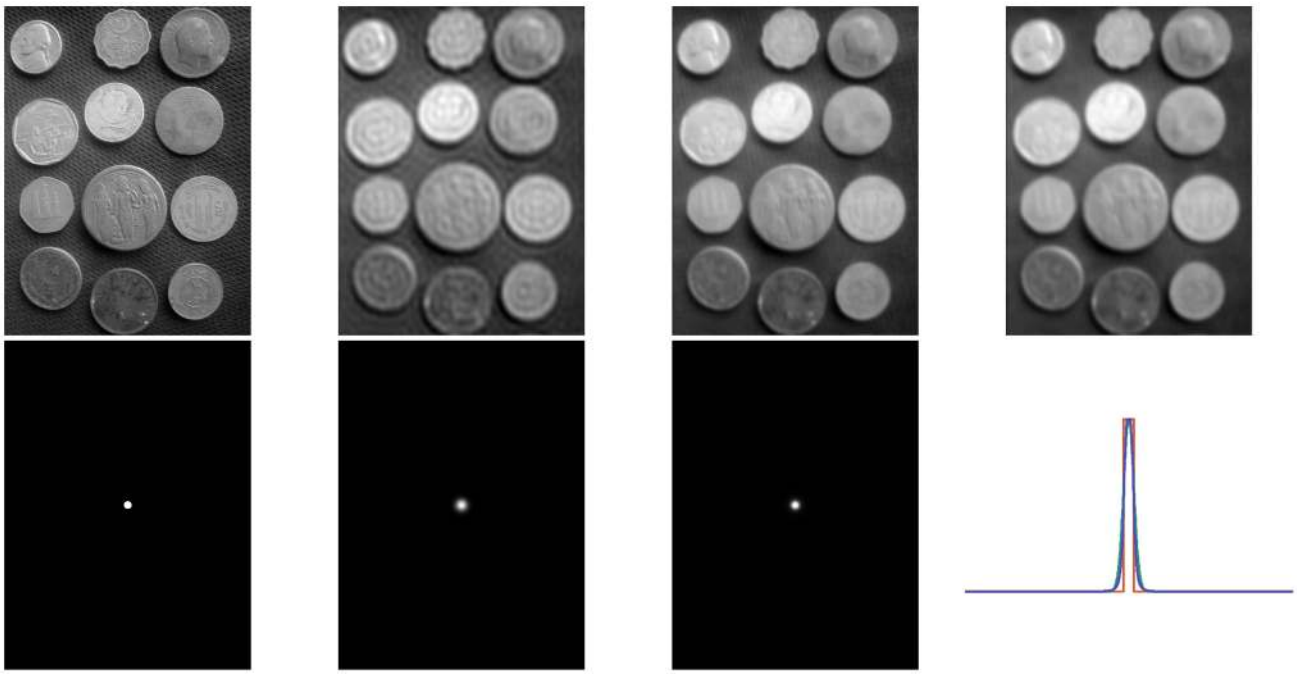


Figure 5.13: Comparação entre filtros passa-baixa: Ideal (D =30), Gaussiano (D =30) e Butterworth (D =30, n=2). Perfis de H(u,v) ao longo de uma linha central e imagens filtradas correspondentes.

```
%%writefile tmp/fig_05_filtros_passa_alta.cpp
#define MM_OUT "tmp/fig_05_filtros_passa_alta.png"
#include <opencv2/opencv.hpp>
#include "morph.hpp"
#include <vector>
#include <string>
#include <filesystem>

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_20.png");
// [pdi:state-io:end]

    //| label: fig-05-filtros-passa-alta
    //| fig-cap: "High-pass Gaussian filter. (a) Original; (b) High-pass filter (D =30) -
    edges of coins and textured background are enhanced."

    // High-pass filter = complement of Gaussian low-pass (1 - H_gauss),
    // built with mm.gaussFilter(..., highpass=True) and applied via FFT.
    int M = img_gray.h;
    int N = img_gray.w;
    double D0 = 30;
    cv::Mat H_alta = mm::gaussFilter(M, N, D0, true);
    mm::Image img_alta = mm::freqFilter(img_gray, H_alta);

    mm::show(
        std::vector<mm::Image>{img_gray, img_alta},
        MM_OUT,
        std::vector<std::string>{"Original", "High-pass Gaussian ($D_0=30$)"},
        2
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_gray, "tmp/fig_05_filtros_passa_alta_0.png");
mm::write(img_alta, "tmp/fig_05_filtros_passa_alta_1.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_05_filtros_passa_alta.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_filtros_passa_alta.cpp
-o tmp/fig_05_filtros_passa_alta -lopencv_stitching -lopencv_alphamat -lopencv_aruco
-lopencv_barcode -lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_filtros_passa_alta \
  && test -f "tmp/fig_05_filtros_passa_alta.png" \
  || echo "  mm::show não gravou tmp/fig_05_filtros_passa_alta.png"
```

```
[1] Original
[2] High-pass Gaussian ($D_0=30$)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_05_filtros_passa_alta_0.png"),
            mm.read("tmp/fig_05_filtros_passa_alta_1.png"),
        ],
        titles=[
            'Original',
            'Passa-alta Gaussiano ($D_0=30$)',
        ],
        cols=2,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_filtros_passa_alta_0.png (ver a versao Python)")
```

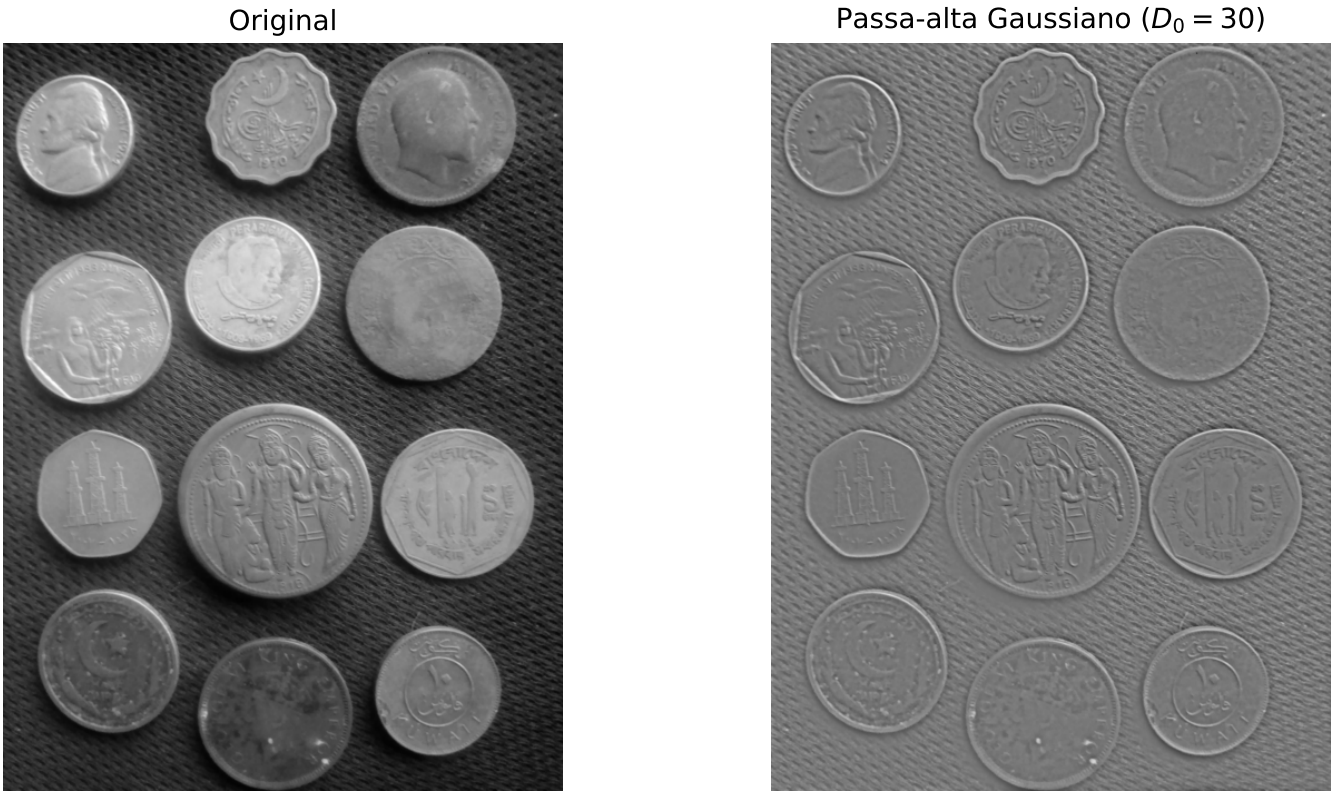


Figure 5.14: Filtro passa-alta Gaussiano. (a) Original; (b) Filtro passa-alta (D =30) - as bordas das moedas e fundo texturizado são realçados.

5.5.3 Periodic Noise Removal

Periodic noise — associated with electrical interference, regular sensor patterns, or scanning artifacts — appears in the Fourier spectrum as **symmetrical point peaks around the center**.

The **notch filter** selectively attenuates these frequencies while preserving the remaining image components. An application example is presented in Figure 5.15.

```
%%writefile tmp/fig_05_ruido_periodico.cpp
#define MM_OUT "tmp/fig_05_ruido_periodico.png"
```

```
//| label: fig-05-ruido-periodico
//| fig-cap: "Remoção de ruído periódico via filtro *notch* no domínio da frequência: (a)
imagem com ruído senoidal, (b) espectro mostrando os picos do ruído, (c) máscara *notch*
centrada nos picos, (d) imagem restaurada."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include "morph.hpp"

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_20.png");
// [pdi:state-io:end]

    // Ruído senoidal 2D -> espectro -> máscara notch nos 4 picos simétricos -> restauração.
    int h_img = img_gray.h;
    int w_img = img_gray.w;

    // Create X, Y meshgrid
    cv::Mat X(h_img, w_img, CV_64F);
    cv::Mat Y(h_img, w_img, CV_64F);
    for(int y = 0; y < h_img; y++) {
        for(int x = 0; x < w_img; x++) {
            X.at<double>(y, x) = x;
            Y.at<double>(y, x) = y;
        }
    }

    int u0 = 20, v0 = 20;  // frequências exatas do ruído

    // Generate sinusoidal noise
    cv::Mat ruido(h_img, w_img, CV_64F);
    cv::Mat img_ruidosa_f(h_img, w_img, CV_64F);

    for(int y = 0; y < h_img; y++) {
        for(int x = 0; x < w_img; x++) {
            double val = 40.0 * std::sin(2.0 * M_PI * (u0 * x / (double)w_img + v0 * y /
(double)h_img));
            ruido.at<double>(y, x) = val;
            double v = (double)img_gray.data[y * w_img + x] + val;
            img_ruidosa_f.at<double>(y, x) = std::min(255.0, std::max(0.0, v));
        }
    }

    mm::Image img_ruidosa(img_ruidosa_f);

    // Espectro de magnitude (log) da imagem ruidosa
    mm::Image mag_vis = mm::spectrumMag(img_ruidosa);

    // Máscara notch: 1.0 em todo o plano, disco de 0.0 em cada um dos 4 picos
    cv::Mat mascara(h_img, w_img, CV_64F, cv::Scalar(1.0));
    int r_notch = 8;
    int cy = h_img / 2, cx = w_img / 2;

    for(auto [dy, dx] : {std::make_pair(v0, u0), std::make_pair(-v0, -u0),
                         std::make_pair(v0, -u0), std::make_pair(-v0, u0)}) {
```

```
        for(int y = 0; y < h_img; y++) {
            for(int x = 0; x < w_img; x++) {
                double dist = std::sqrt(std::pow(y - (cy + dy), 2) + std::pow(x - (cx + dx),
2));
                if(dist <= r_notch) {
                    mascara.at<double>(y, x) = 0.0;
                }
            }
        }
    }

    cv::Mat mascara_8u;
    mascara.convertTo(mascara_8u, CV_8U, 255.0);
    mm::Image mascara_vis(mascara_8u);

    // Filtragem: aplica a máscara centrada como filtro no domínio da frequência
    mm::Image img_rest_vis = mm::freqFilter(img_ruidosa, mascara);

    // Convert img_gray and img_rest_vis to cv::Mat for PSNR
    cv::Mat img_gray_cv = img_gray;
    cv::Mat img_rest_cv = img_rest_vis;

    double psnr = cv::PSNR(img_gray_cv, img_rest_cv);
    std::cout << "PSNR (original vs restaurada): " << psnr << " dB" << std::endl;

    std::vector<mm::Image> results = {img_ruidosa, mag_vis, mascara_vis, img_rest_vis};
    std::vector<std::string> titles = {
        "Com ruído periódico",
        "Espectro (log)",
        "Máscara notch",
        "Restaurada (PSNR=" + std::to_string(psnr).substr(0, 4) + " dB)"
    };

    mm::show(results, MM_OUT, titles, 4);

    return 0;
}
```

```
Overwriting tmp/fig_05_ruido_periodico.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_ruido_periodico.cpp -o
tmp/fig_05_ruido_periodico -lopencv_stitching -lopencv_alphamat -lopencv_aruco
-lopencv_barcode -lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_ruido_periodico \
  && test -f "tmp/fig_05_ruido_periodico.png" \
  || echo "  mm::show não gravou tmp/fig_05_ruido_periodico.png"
```

```
PSNR (original vs restaurada): 33.0718 dB
[1] Com ruído periódico
[2] Espectro (log)
[3] Máscara notch
```

```
[4] Restaurada (PSNR=33.0 dB)
```

```
try:
    mm.show(mm.read("tmp/fig_05_ruido_periodico.png"), figsize=(16, 4))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_ruido_periodico.png (ver a versao Python)")
```

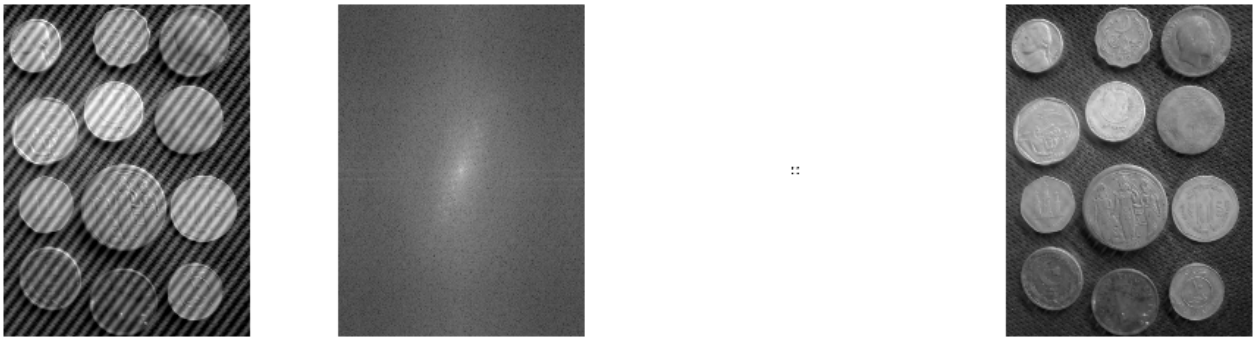


Figure 5.15: Remoção de ruído periódico via filtro *notch* no domínio da frequência: (a) imagem com ruído senoidal, (b) espectro mostrando os picos do ruído, (c) máscara *notch* centrada nos picos, (d) imagem restaurada.

📌 Synthesis — Spectral Filters

Filter	Visual effect	Artifact	Use
Ideal low-pass	Intense smoothing	*Ringing*	Illustrative
Gaussian low-pass	Gentle smoothing	No *ringing*	General smoothing
Butterworth low-pass	Controlled smoothing	*Ringing* (high orders)	Trade-off between smoothing and selectivity
High-pass	Edge enhancement	Noise amplification	Contour detection
Notch	Selective frequency removal	Possible local distortions	Periodic noise removal

The design of filters in the frequency domain consists of defining spectral masks. However, effects in the spatial domain, such as *ringing* and blurring, emerge directly from these choices in the spectrum.

5.6 *Wavelets* and Multiresolution

The Fourier Transform decomposes the signal into **global** frequencies: each coefficient $F(u, v)$ receives contributions from the entire image, with no explicit information about the spatial localization of those frequencies. Thus, localized structures, such as edges, are represented in a distributed manner across the spectrum.

***Wavelets* (ondaletas)** overcome this limitation by using basis functions **localized in space**, which can be translated and scaled. These functions have **compact support**, that is, they are nonzero only in a finite region of the domain, allowing a simultaneous representation in terms of **frequency and spatial localization**.

5.6.1 The Limit of the Fourier Transform: Spatial Localization

The Fourier Transform accurately describes **which frequencies are present** in a signal, but it does not explicitly represent **where those frequencies occur in space**.

In the experiment presented in Figure 5.16, two images with structures located at different positions produce nearly identical magnitude spectra. This occurs because the Fourier representation is global: each coefficient receives contributions from the entire image.

As a result, the magnitude spectrum does not explicitly represent the location of edges or other structures, only the distribution of the frequencies present. This limitation motivated the development of multiresolution representations, such as the Discrete *Wavelet* Transform (DWT), capable of simultaneously describing the frequency and spatial localization of image structures.

```
%%writefile tmp/fig_05_fracasso_fourier.cpp
#define MM_OUT "tmp/fig_05_fracasso_fourier.png"
//| label: fig-05-fracasso-fourier
//| fig-cap: "Fourier global é cego para a posição. Os espectros não dizem onde as bordas
estão."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include "morph.hpp"
#include <filesystem>

// Helper to shift the zero-frequency component to the center (fftshift for 2D)
void fftshift2D(cv::Mat& src, cv::Mat& dst) {
    int cx = src.cols / 2;
    int cy = src.rows / 2;
    cv::Mat q0(src, cv::Rect(0, 0, cx, cy));   // Top-Left
    cv::Mat q1(src, cv::Rect(cx, 0, src.cols - cx, cy)); // Top-Right
    cv::Mat q2(src, cv::Rect(0, cy, cx, src.rows - cy)); // Bottom-Left
    cv::Mat q3(src, cv::Rect(cx, cy, src.cols - cx, src.rows - cy)); // Bottom-Right

    cv::Mat tmp;
    q0.copyTo(tmp);
    q3.copyTo(q0);
    tmp.copyTo(q3);

    q1.copyTo(tmp);
    q2.copyTo(q1);
    tmp.copyTo(q2);

    dst = src.clone();
}

int main() {
    // Create signal images
    cv::Mat img_sinal1 = cv::Mat::zeros(128, 128, CV_64F);
    img_sinal1(cv::Rect(20, 0, 5, 128)).setTo(1.0);  // Vertical bar at x=20:25
    img_sinal1(cv::Rect(0, 100, 128, 5)).setTo(1.0); // Horizontal bar at y=100:105

    cv::Mat img_sinal2 = cv::Mat::zeros(128, 128, CV_64F);
    img_sinal2(cv::Rect(90, 0, 5, 128)).setTo(1.0);  // Vertical bar at x=90:95
    img_sinal2(cv::Rect(0, 30, 128, 5)).setTo(1.0);  // Horizontal bar at y=30:35

    // Compute Fourier transforms
    cv::Mat fft1, fft2;
```

```
    cv::dft(img_sinal1, fft1, cv::DFT_COMPLEX_OUTPUT);
    cv::dft(img_sinal2, fft2, cv::DFT_COMPLEX_OUTPUT);

    // Split into real/imaginary parts
    std::vector<cv::Mat> planes1, planes2;
    cv::split(fft1, planes1);
    cv::split(fft2, planes2);

    // Compute magnitude with shift
    cv::Mat mag_tmp1, mag_tmp2;
    cv::magnitude(planes1[0], planes1[1], mag_tmp1);
    cv::magnitude(planes2[0], planes2[1], mag_tmp2);

    // Shift and apply log(1+|F|)
    cv::Mat mag_shifted1, mag_shifted2;
    fftshift2D(mag_tmp1, mag_shifted1);
    fftshift2D(mag_tmp2, mag_shifted2);

    cv::Mat mag1, mag2;
    cv::log(mag_shifted1 + 1.0, mag1);
    cv::log(mag_shifted2 + 1.0, mag2);

    // Normalize for display (convert to 8-bit)
    cv::Mat disp1, disp2;
    cv::normalize(img_sinal1, disp1, 0, 255, cv::NORM_MINMAX, CV_8U);
    cv::normalize(img_sinal2, disp2, 0, 255, cv::NORM_MINMAX, CV_8U);
    cv::normalize(mag1, mag1, 0, 255, cv::NORM_MINMAX, CV_8U);
    cv::normalize(mag2, mag2, 0, 255, cv::NORM_MINMAX, CV_8U);

    // Display results
    std::vector<mm::Image> imgs = {mm::Image(disp1), mm::Image(mag1), mm::Image(disp2),
mm::Image(mag2)};
    std::vector<std::string> titles = {"Sinal A", "Espectro A", "Sinal B (Deslocado)",
"Espectro B"};
    mm::show(imgs, MM_OUT, titles, 4);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_sinal1, "tmp/fig_05_fracasso_fourier_0.png");
mm::write(mag1, "tmp/fig_05_fracasso_fourier_1.png");
mm::write(img_sinal2, "tmp/fig_05_fracasso_fourier_2.png");
mm::write(mag2, "tmp/fig_05_fracasso_fourier_3.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_05_fracasso_fourier.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_fracasso_fourier.cpp -o
tmp/fig_05_fracasso_fourier -lopencv_stitching -lopencv_alphamat -lopencv_aruco
-lopencv_barcode -lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_fracasso_fourier \
  && test -f "tmp/fig_05_fracasso_fourier.png" \
  || echo "  mm::show não gravou tmp/fig_05_fracasso_fourier.png"
```

```
[1] Sinal A
[2] Espectro A
[3] Sinal B (Deslocado)
[4] Espectro B
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_05_fracasso_fourier_0.png"),
            mm.read("tmp/fig_05_fracasso_fourier_1.png"),
            mm.read("tmp/fig_05_fracasso_fourier_2.png"),
            mm.read("tmp/fig_05_fracasso_fourier_3.png"),
        ],
        titles=[
            'Sinal A',
            'Espectro A',
            'Sinal B (Deslocado)',
            'Espectro B',
        ],
        cols=4,
        figsize=(14, 4),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_fracasso_fourier_0.png (ver a versao Python)")
```

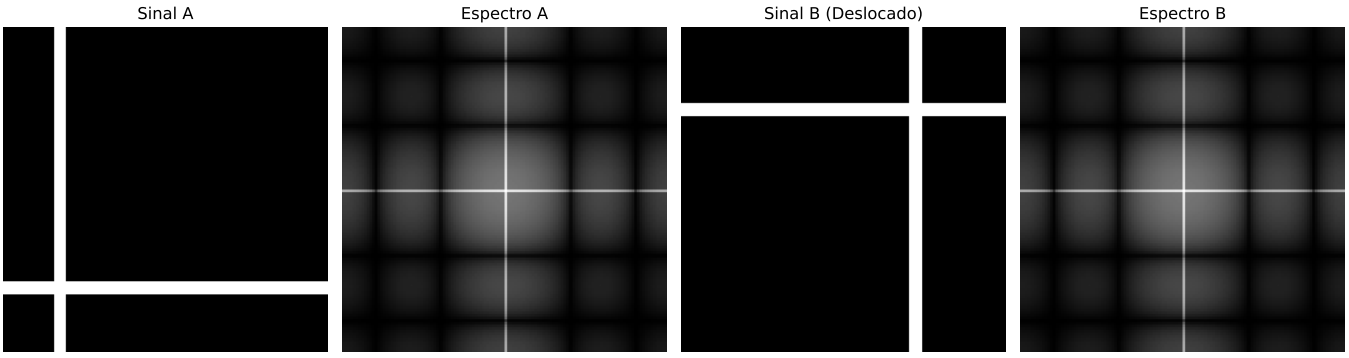


Figure 5.16: Fourier global é cego para a posição. Os espectros não dizem onde as bordas estão.

5.6.2 2D Discrete Wavelet Transform

The Discrete Wavelet Transform (DWT) applies, separately in the horizontal and vertical directions, two complementary filters: a **low-pass** h (approximation) and a **high-pass** g (details), followed by downsampling by a factor of 2 in each dimension. This process produces four subbands, whose names indicate the combination of filters applied in each direction (L = *Low-pass*; H = *High-pass*). The characteristics of each subband are summarized in Table 5.3.

$$\mathrm{DWT}(f) = \{ \underbrace{\mathrm{LL}}_{\text{aprox.}}, \underbrace{\mathrm{LH}}_{\text{horizontal details}}, \underbrace{\mathrm{HL}}_{\text{vertical details}}, \underbrace{\mathrm{HH}}_{\text{diagonal details}} \}.$$

Table 5.3: Subbands produced by the 2D Discrete Wavelet Transform (DWT), indicating the filters applied in each direction and the predominant content of each component.

Subband	Filters applied	Visual content
LL	low × low	Image approximation (smoothed and reduced version)
LH	low × high	Horizontal edges and vertical variations
HL	high × low	Vertical edges and horizontal variations
HH	high × high	Diagonal details and textures

The decomposition can be applied recursively to the LL subband, generating a multiresolution representation. After J levels, a structure with $3J + 1$ subbands is obtained, where each new level reduces the resolution of the approximation component.

> **Connection with CNNs**
>
> The multiresolution decomposition of wavelets has a conceptual relationship with the hierarchical representations used in convolutional neural networks (CNNs). In both cases, successive stages of filtering and resolution reduction produce increasingly abstract descriptions of the image. However, **wavelets use mathematically defined and reconstructible filters**, whereas **CNNs learn their filters during training**.

5.6.3 Wavelet Families

Different wavelet families present distinct trade-offs between **spatial support**, smoothness, and compression capability. **Support** corresponds to the extent of the wavelet function in the spatial domain: the smaller the support, the more localized the function; the larger it is, the smoother its representation tends to be, albeit at a higher computational cost. Table 5.4 compares some of the most commonly used families.

Table 5.4: Comparison among wavelet families, highlighting support length, number of vanishing moments, symmetry, and typical applications.

Wavelet	Support length	Vanishing moments	Symmetry	Typical use
Haar	2	1	Asymmetric	Introduction and basic analysis
Daubechies db4	8	4	Asymmetric	Compression and general analysis
Symlet sym4	8	4	Nearly symmetric	Signal reconstruction

Wavelet	Support length	Vanishing moments	Symmetry	Typical use
Biorthogonal 5/3	5/3	2/2	Symmetric	Lossless JPEG 2000
Biorthogonal 9/7	9/7	4/4	Symmetric	Lossy JPEG 2000

Vanishing moments measure the wavelet's ability to represent smooth regions of the image with few nonzero coefficients. A wavelet with p vanishing moments exactly annihilates polynomials of degree up to $p-1$. Consequently, the greater the number of vanishing moments, the greater the compression efficiency tends to be in homogeneous regions, although this generally implies functions with longer support.

Figure 5.17 presents the basis functions (wavelets) $\psi(t)$ in the spatial domain. These functions have **compact support**, that is, they are nonzero only in a finite region of the domain, unlike the sinusoids of the Fourier Transform, which extend over the entire domain.

```
%%writefile tmp/fig_05_wavelet_functions.cpp
#define MM_OUT "tmp/fig_05_wavelet_functions.png"
//| label: fig-05-wavelet-functions
//| fig-cap: "Funções da *Wavelet* ( ). Note como elas rapidamente decaem para zero (suporte
compacto), ao contrário das senoides infinitas de Fourier."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
#include <string>
#include <vector>

#define MM_USE_OPENCV
#include "morph.hpp"
#include <filesystem>

// psi via mm.wavefun (algoritmo em cascata) - suporte compacto: decai a zero.
int main() {
    std::vector<double> x_h, phi_h, psi_h;
    mm::wavefun("haar", 6, x_h, phi_h, psi_h);

    std::vector<double> x_d, phi_d, psi_d;
    mm::wavefun("db4", 6, x_d, phi_d, psi_d);

    mm::Image chart_h = mm::lineChart(
        std::vector<std::vector<double>>{x_h},
        std::vector<std::vector<double>>{psi_h},
        std::vector<std::string>{"psi Haar"},
        std::vector<cv::Scalar>{cv::Scalar(40, 60, 180)},
        "Ondaleta Haar (psi)", "t");

    mm::Image chart_d = mm::lineChart(
        std::vector<std::vector<double>>{x_d},
        std::vector<std::vector<double>>{psi_d},
        std::vector<std::string>{"psi Daubechies 4"},
        std::vector<cv::Scalar>{cv::Scalar(40, 140, 60)},
        "Ondaleta Daubechies 4 (psi)", "t");

    mm::show(std::vector<mm::Image>{chart_h, chart_d},
             MM_OUT,
             std::vector<std::string>{"Haar", "Daubechies 4"},
             2);
```

```
// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(chart_h, "tmp/fig_05_wavelet_functions_0.png");
mm::write(chart_d, "tmp/fig_05_wavelet_functions_1.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_05_wavelet_functions.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_wavelet_functions.cpp -o
tmp/fig_05_wavelet_functions -lopencv_stitching -lopencv_alphamat -lopencv_aruco
-lopencv_barcode -lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_wavelet_functions \
  && test -f "tmp/fig_05_wavelet_functions.png" \
  || echo "  mm::show não gravou tmp/fig_05_wavelet_functions.png"
```

```
tmp/fig_05_wavelet_functions.cpp:11: warning: "MM_USE_OPENCV" redefined
   11 | #define MM_USE_OPENCV
      |
<command-line>: note: this is the location of the previous definition
[1] Haar
[2] Daubechies 4
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_05_wavelet_functions_0.png"),
            mm.read("tmp/fig_05_wavelet_functions_1.png"),
        ],
        titles=[
            'Haar',
            'Daubechies 4',
        ],
        cols=2,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_wavelet_functions_0.png (ver a versao Python)")
```

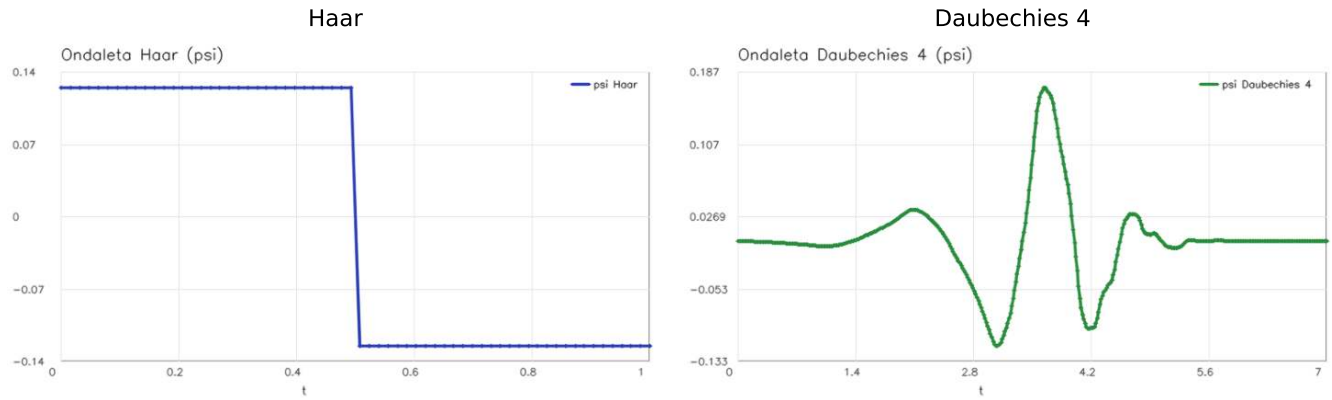


Figure 5.17: Funções da *Wavelet* ( ). Note como elas rapidamente decaem para zero (suporte compacto), ao contrário das senoides infinitas de Fourier.

The diagram in Figure 5.18 illustrates the multiresolution analysis performed by the DWT, in which the approximation subband (LL) is successively decomposed, forming a hierarchical representation with two levels.

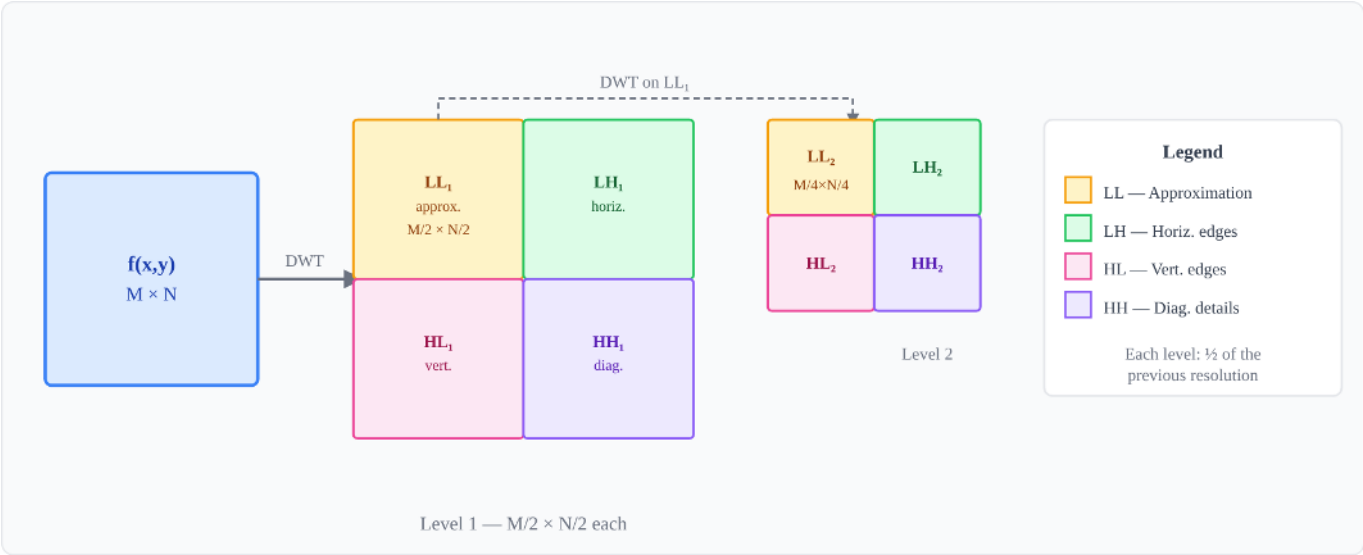


Figure 5.18: Diagram of the 2D wavelet decomposition at two levels.

The simulator in Figure 5.19 allows for interactive exploration of the 2D Discrete *Wavelet* Transform (DWT) using the Haar *wavelet*. The subband decomposition highlights the separation between the approximation component and the detail components of the image.

The different input patterns allow observing the directional behavior of the filters. In images with horizontal and vertical edges, the LH and HL subbands highlight, respectively, the vertical and horizontal intensity variations. In regions of smooth variation, most of the energy is concentrated in the LL approximation subband, while the detail subbands exhibit coefficients close to zero.

Multiresolution analysis can also be observed by increasing the number of decomposition levels. In this case, only the LL_1 subband is further decomposed, giving rise to the LL_2, LH_2, HL_2, and HH_2 subbands, which form the second level of the hierarchical representation.

In patterns composed of large homogeneous regions, such as a smooth gradient or a checkerboard made of large blocks, the energy remains predominantly concentrated in the LL subband. In the gradient, this occurs because the differences between neighboring pixels are small. In the checkerboard, in turn, the pixels have practically the same intensity within each block, so that only the boundaries between blocks produce nonzero

coefficients in the detail subbands. Since these boundaries occupy only a small fraction of the image, their contribution to the total energy remains reduced.

To enable the visual analysis of these subtle variations, the simulator incorporates a contrast gain control for the details (ranging from 1 to 8). This parameter functions as a linear amplification factor applied exclusively to the coefficients of the detail subbands (LH, HL, and HH) before their on-screen rendering. In scenarios of smooth transition (such as the gradient) or local uniformity (such as the interior of the checkerboard blocks), the numerical differences calculated by the Haar high-pass filter result in coefficients very close to zero, which would render the corresponding quadrants dark and imperceptible to the naked eye. By multiplying these values by the gain, the simulator visually recovers the hidden high-frequency structures and enhances the orientation of the remaining edges.

The energy-per-subband plot quantifies this distribution between the approximation component and the detail components, demonstrating that the visual gain does not alter the original energy metric. In natural images, most of the energy is concentrated in the LL subband, while the LH, HL, and HH subbands mainly represent edges, textures, and other local intensity variations.

5.6.4 Multiresolution Analysis with the 2D DWT

The 2D Discrete Wavelet Transform (DWT) decomposes an image into approximation and detail components, organized hierarchically across different scales and orientations. As detail subbands in natural images often exhibit low-contrast coefficients, the practical examples below use a synthetic geometric pattern generated in Python. This approach replicates the behavior of the simulator in Figure 5.19, making the effects of spatial filtering and multiresolution decomposition visually explicit.

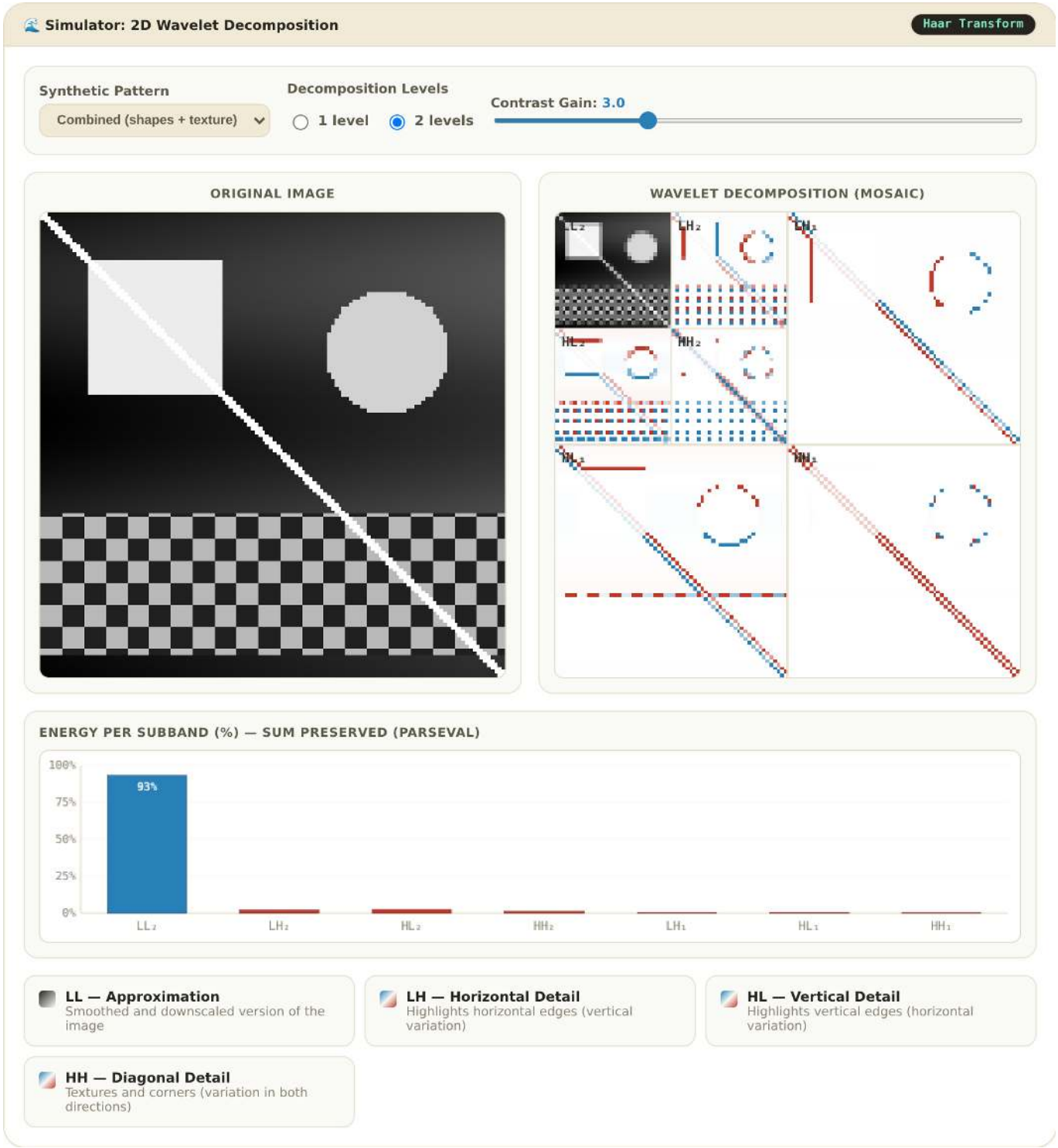
5.6.4.1 Multi-Level Mosaic Decomposition

Figure 5.20 illustrates the hierarchical structure of the two-level DWT using the Haar *wavelet*. The process is based on the combined application of low-pass and high-pass filters in the horizontal and vertical directions, followed by subsampling by a factor of 2.

In the first level, the original image yields the approximation subband (LL_1) and the horizontal (LH_1), vertical (HL_1), and diagonal (HH_1) detail components. In multiresolution analysis, the LL_1 subband is again filtered and subsampled, generating the second decomposition level (LL_2, LH_2, HL_2, and HH_2).

To enable the visual interpretation of the detail components, the code extracts the absolute value of their coefficients and applies linear (*min-max*) normalization to occupy the entire dynamic range of gray levels [0, 255]. This operation transforms homogeneous regions (zero coefficients) into black and highlights in white the edges and textures extracted at each scale and orientation.

```
%%writefile tmp/fig_05_dwt_subbandas.cpp
#define MM_OUT "tmp/fig_05_dwt_subbandas.png"
#include <opencv2/opencv.hpp>
#include <iostream>
#include <string>
#include <vector>
#include <cmath>
#include "morph.hpp"

//| label: fig-05-dwt-subbandas
//| fig-cap: "Decomposição *wavelet* 2D de 2 níveis com *wavelet* Haar: subbandas LL, LH, HL,
HH em cada nível. As subbandas de detalhe revelam estruturas orientadas em diferentes escalas
utilizando um padrão sintético."
//| echo: true
//| output: true

//   Geração da Imagem Sintética (Mesmo padrão 'combined' do simulador)
mm::Image gerar_imagem_sintetica(int N = 256) {
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap05/sim-05-wavelet.html

Figure 5.19: Simulation of 2D *wavelet* decomposition.

```
    cv::Mat img = cv::Mat::zeros(N, N, CV_64F);
    for (int y = 0; y < N; y++) {
        for (int x = 0; x < N; x++) {
            double v = 55 + 35 * ((double)x / N) + 15 * std::sin((double)y / 24);
            // Quadrado
            if (24 < x && x < 100 && 24 < y && y < 100) {
                v = 225;
            }
            // Círculo
            int cx = 190, cy = 76, r = 34;
            if ((x - cx) * (x - cx) + (y - cy) * (y - cy) < r * r) {
                v = 205;
            }
            // Textura periódica (inferior)
            if (y > 164 && y < 244) {
                int p = 12;
                v = (((x / p + y / p) % 2) == 0) ? 185 : 65;
            }
            // Linha diagonal
            if (std::abs(x - y) < 4) {
                v = 240;
            }
            img.at<double>(y, x) = std::max(0.0, std::min(255.0, v));
        }
    }
    cv::Mat img_uint8;
    img.convertTo(img_uint8, CV_8U);
    mm::Image out = img_uint8;
    return out;
}

// Normaliza subbanda para visualização [0,255]
mm::Image sb_vis(const cv::Mat& sb) {
    cv::Mat sb_abs;
    cv::magnitude(sb, cv::Mat::zeros(sb.size(), sb.type()), sb_abs);
    cv::Mat sb_norm;
    cv::normalize(sb_abs, sb_norm, 0, 255, cv::NORM_MINMAX, CV_8U);
    mm::Image out = sb_norm;
    return out;
}

int main() {
    // Substitui a imagem escura de moedas pelo padrão sintético claro
    mm::Image img_gray_mm = gerar_imagem_sintetica(256);
    cv::Mat img_gray(img_gray_mm.h, img_gray_mm.w, CV_8UC1, img_gray_mm.data.data());

    //   Decomposição wavelet 2 níveis
    std::string wavelet = "haar";
    cv::Mat img_float;
    img_gray.convertTo(img_float, CV_64F);

    // Nível 1
    mm::Subbands coefs1 = mm::dwt2(img_float, wavelet);
    cv::Mat LL1 = coefs1.LL;
    cv::Mat LH1 = coefs1.LH;
    cv::Mat HL1 = coefs1.HL;
    cv::Mat HH1 = coefs1.HH;

    // Nível 2 (aplicado sobre LL1)
    mm::Subbands coefs2 = mm::dwt2(LL1, wavelet);
    cv::Mat LL2 = coefs2.LL;
```

```
    cv::Mat LH2 = coefs2.LH;
    cv::Mat HL2 = coefs2.HL;
    cv::Mat HH2 = coefs2.HH;

    std::cout << "Forma original     : " << img_gray.rows << "x" << img_gray.cols <<
std::endl;
    std::cout << "LL1 (nível 1)      : " << LL1.rows << "x" << LL1.cols << "  |  LH1/HL1/HH1:
" << LH1.rows << "x" << LH1.cols << std::endl;
    std::cout << "LL2 (nível 2)      : " << LL2.rows << "x" << LL2.cols << "    |
LH2/HL2/HH2: " << LH2.rows << "x" << LH2.cols << std::endl;

    std::vector<mm::Image> imgs_dwt = {
        img_gray_mm, sb_vis(LL1), sb_vis(LH1), sb_vis(HL1), sb_vis(HH1),
        sb_vis(LL2), sb_vis(LH2), sb_vis(HL2), sb_vis(HH2)
    };
    std::vector<std::string> titles_dwt = {
        "Original",
        "LL  (aprox.)", "LH  (horiz.)", "HL  (vert.)", "HH  (diag.)",
        "LL  (aprox.)", "LH  (horiz.)", "HL  (vert.)", "HH  (diag.)"
    };

    mm::show(imgs_dwt, MM_OUT, titles_dwt, 5);

    return 0;
}
```

```
Overwriting tmp/fig_05_dwt_subbandas.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_dwt_subbandas.cpp -o
tmp/fig_05_dwt_subbandas -lopencv_stitching -lopencv_alphamat -lopencv_aruco -lopencv_barcode
-lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_dwt_subbandas \
  && test -f "tmp/fig_05_dwt_subbandas.png" \
  || echo "  mm::show não gravou tmp/fig_05_dwt_subbandas.png"
```

```
Forma original     : 256x256
LL1 (nível 1)      : 128x128  |  LH1/HL1/HH1: 128x128
LL2 (nível 2)      : 64x64    |  LH2/HL2/HH2: 64x64
[1] Original
[2] LL  (aprox.)
[3] LH  (horiz.)
[4] HL  (vert.)
[5] HH  (diag.)
[6] LL  (aprox.)
[7] LH  (horiz.)
[8] HL  (vert.)
[9] HH  (diag.)
```

```
try:
    mm.show(mm.read("tmp/fig_05_dwt_subbandas.png"), figsize=(16, 7))
```

```
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_dwt_subbandas.png (ver a versao Python)")
```

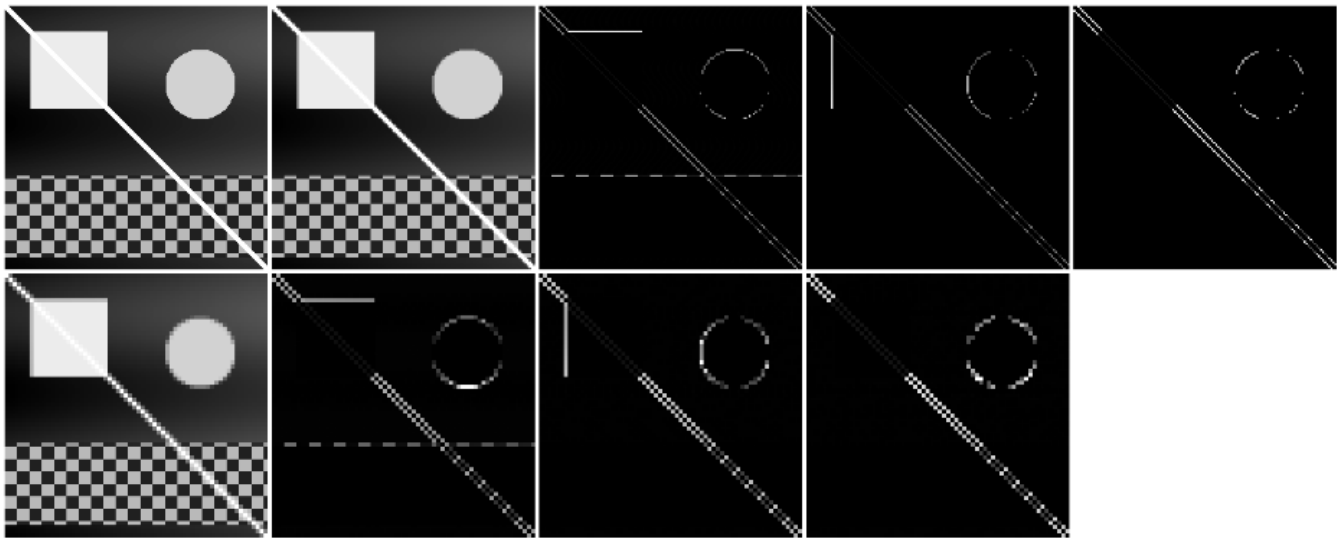


Figure 5.20: Decomposição *wavelet* 2D de 2 níveis com *wavelet* Haar: subbandas LL, LH, HL, HH em cada nível. As subbandas de detalhe revelam estruturas orientadas em diferentes escalas utilizando um padrão sintético.

5.6.4.2 The Trade-off Between Localization and Smoothness

The choice of the basis function (wavelet) directly influences how image features are distributed and encoded by the DWT coefficients. Figure 5.21 compares the practical results obtained by applying four distinct families to the synthetic geometric pattern: `haar`, `db4`, `sym4`, and `bior2.2`.

Due to its short support and step-function shape, the Haar wavelet produces coefficients that are highly localized at spatial discontinuities, generating thin, sharp edges in the detail subbands. In contrast, families such as Daubechies (`db4`) and Symlets (`sym4`), which exhibit larger support (longer filters) and a greater number of vanishing moments, yield smoother, more distributed responses around transitions, which can introduce slight oscillations or blurring at abrupt boundaries.

This behavior highlights the classic trade-off in multiresolution analysis: smaller supports favor precise spatial localization of edges, while larger supports and a greater number of vanishing moments tend to produce sparser, smoother representations. This smoothness and ability to attenuate high frequencies ensure greater efficiency in energy compaction—fundamental characteristics for data compression and denoising applications.

```
%%writefile tmp/fig_05_dwt_wavelets.cpp
#define MM_OUT "tmp/fig_05_dwt_wavelets.png"
//| label: fig-05-dwt-wavelets
//| fig-cap: "Comparison between wavelet families: Haar, db4, sym4 and bior2.2. LL1 subband
(approximation) and HH1 (diagonal) for each choice, illustrating the trade-off between
compaction and smoothness based on the synthetic pattern."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <cmath>
#include "morph.hpp"
```

```cpp
// Helper to visualize a subband (normalize to 0-255)
static cv::Mat sb_vis(const cv::Mat& subband) {
    cv::Mat vis;
    cv::normalize(subband, vis, 0, 255, cv::NORM_MINMAX, CV_8U);
    return vis;
}

int main() {
    // Garante que img_gray e img_float utilizem o mesmo padrão sintético claro
    cv::Mat img_gray;
    // Note: gerar_imagem_sintetica from previous cell would be available
    // Fallback: generate the synthetic image
    int N = 256;
    cv::Mat img_gen(N, N, CV_8UC1);
    for (int y = 0; y < N; y++) {
        for (int x = 0; x < N; x++) {
            double v = 55.0 + 35.0 * ((double)x / N) + 15.0 * std::sin(y / 24.0);
            if (24 < x && x < 100 && 24 < y && y < 100) v = 225.0;
            double cx = 190.0, cy = 76.0, r = 34.0;
            double dx = x - cx, dy = y - cy;
            if (dx*dx + dy*dy < r*r) v = 205.0;
            if (y > 164 && y < 244) {
                int p = 12;
                v = (((x / p + y / p) % 2) == 0) ? 185.0 : 65.0;
            }
            if (std::abs(x - y) < 4) v = 240.0;
            v = std::max(0.0, std::min(255.0, v));
            img_gen.at<uchar>(y, x) = (uchar)v;
        }
    }
    img_gray = img_gen;

    cv::Mat img_float;
    img_gray.convertTo(img_float, CV_64F);

    std::vector<std::string> wavelets_comp = {"haar", "db4", "sym4", "bior2.2"};
    std::vector<cv::Mat> imgs_comp;
    std::vector<std::string> titles_comp;

    for (const auto& wname : wavelets_comp) {
        // pywt.dwt2(img_float, wname) -> LL, (LH, HL, HH)
        auto sub = mm::dwt2(img_float, wname);
        imgs_comp.push_back(sb_vis(sub.LL));
        imgs_comp.push_back(sb_vis(sub.HH));
        titles_comp.push_back(wname + " - LL1");
        titles_comp.push_back(wname + " - HH1");
    }

    // Build a single image grid for display
    int cols = 4, rows = (int)imgs_comp.size() / cols;
    int cell_h = imgs_comp[0].rows, cell_w = imgs_comp[0].cols;
    // Assume all subbands same size; resize to a common display size
    int disp_w = 200, disp_h = 200;
    cv::Mat grid(rows * (disp_h + 30), cols * (disp_w + 10), CV_8UC1, cv::Scalar(255));

    for (size_t k = 0; k < imgs_comp.size(); k++) {
        cv::Mat disp;
        cv::resize(imgs_comp[k], disp, cv::Size(disp_w, disp_h));
        int rr = (int)k / cols, cc = (int)k % cols;
        int ox = cc * (disp_w + 10) + 5, oy = rr * (disp_h + 30) + 25;
```

```
        disp.copyTo(grid(cv::Rect(ox, oy, disp_w, disp_h)));
        // Add title text
        cv::putText(grid, titles_comp[k], cv::Point(ox, oy - 8),
                    cv::FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(0), 1);
    }

    mm::Image out(grid);
    mm::show(out, MM_OUT);

    return 0;
}
```

```
Overwriting tmp/fig_05_dwt_wavelets.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_dwt_wavelets.cpp -o
tmp/fig_05_dwt_wavelets -lopencv_stitching -lopencv_alphamat -lopencv_aruco -lopencv_barcode
-lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_dwt_wavelets \
  && test -f "tmp/fig_05_dwt_wavelets.png" \
  || echo "  mm::show não gravou tmp/fig_05_dwt_wavelets.png"
```

```
try:
    mm.show(mm.read("tmp/fig_05_dwt_wavelets.png"), figsize=(14, 8))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_dwt_wavelets.png (ver a versao Python)")
```

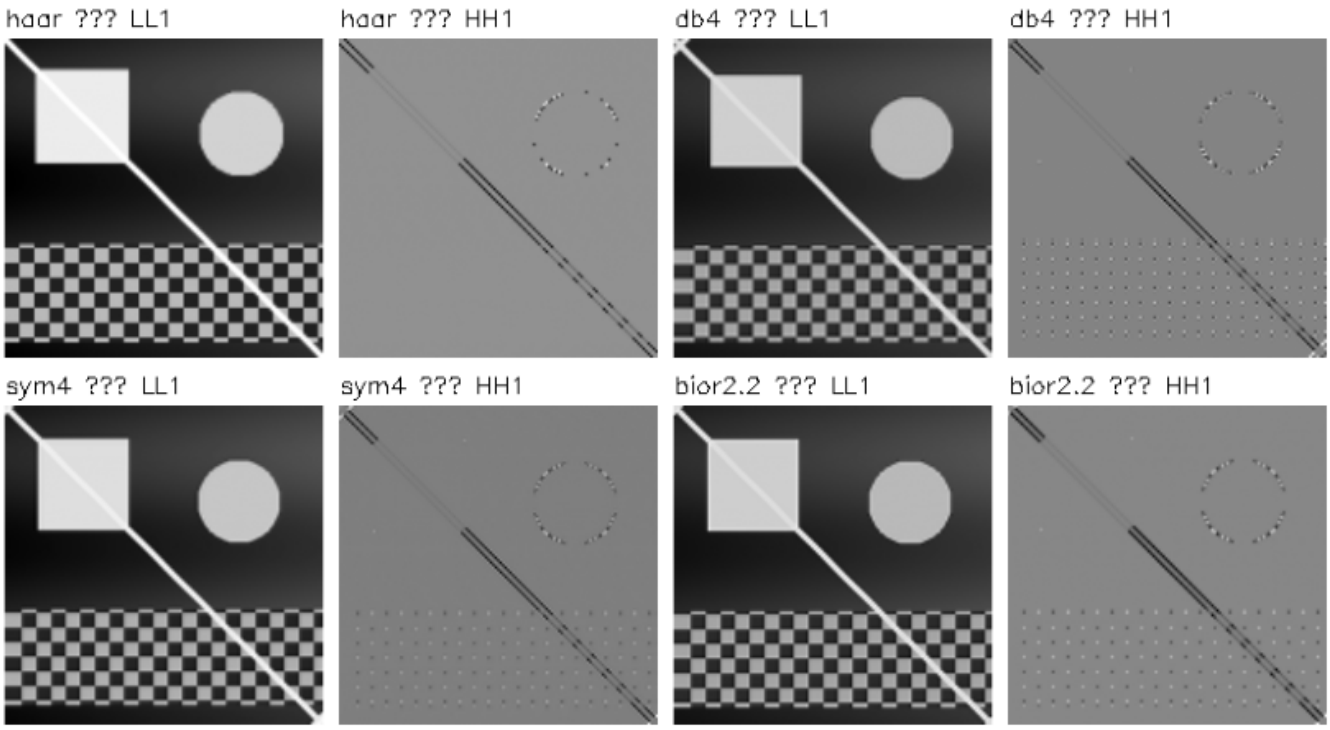


Figure 5.21: Comparação entre famílias de *wavelets*: Haar, db4, sym4 e bior2.2. Subbanda LL (aproximação) e HH (diagonal) para cada escolha, ilustrando o compromisso entre compactação e suavidade com base no padrão sintético.

5.6.4.3 Coefficient Thresholding and Compression

One of the main applications of the Discrete *Wavelet* Transform (DWT) is data compression, driven by the **sparse** representation capability of the coefficients. Figure 5.22 illustrates the effect of hard thresholding, a technique in which detail coefficients with magnitude below a threshold T are entirely zeroed out before the synthesis process carried out by the Inverse Discrete *Wavelet* Transform (IDWT).

As the threshold T is increased, a growing number of high-frequency coefficients are zeroed. Since they concentrate less energy, removing these components considerably reduces the amount of information required to represent the image, while keeping the global approximation component (the deepest LL subband) intact to preserve the macro structure. Visually, this discarding of coefficients manifests itself through the progressive disappearance of fine textures and the smoothing of abrupt intensity transitions.

The fidelity of the reconstructed image relative to the original is quantified by the **Peak Signal-to-Noise Ratio (PSNR)** metric, expressed in decibels (dB). Higher PSNR values indicate less distortion and greater mathematical proximity to the original signal. The practical experiment highlights the gradual decay of PSNR as the thresholding aggressiveness increases, allowing the optimal threshold for the balance between compression and visual degradation to be numerically evaluated.

```
%%writefile tmp/fig_05_dwt_reconstrucao.cpp
#define MM_OUT "tmp/fig_05_dwt_reconstrucao.png"
//| label: fig-05-dwt-reconstrucao
//| fig-cap: "Reconstrução *wavelet* com limiarização de coeficientes (*hard thresholding*): à
medida que o limiar aumenta, mais detalhes são zerados, produzindo imagens progressivamente
mais suaves. Métrica PSNR quantifica a perda de qualidade sobre o padrão sintético."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
#include <vector>
```

```
#include <string>
#include <cmath>
#include <iostream>
#include <algorithm>
#include "morph.hpp"


// Garante que img_gray utilize o mesmo padrão sintético claro
static cv::Mat gerar_imagem_sintetica(int N=256) {
    cv::Mat img = cv::Mat::zeros(N, N, CV_64F);
    for (int y = 0; y < N; y++) {
        for (int x = 0; x < N; x++) {
            double v = 55 + 35 * (static_cast<double>(x) / N) + 15 * std::sin(y / 24.0);
            if (24 < x && x < 100 && 24 < y && y < 100) v = 225;
            int cx = 190, cy = 76, r = 34;
            if ((x - cx)*(x - cx) + (y - cy)*(y - cy) < r*r) v = 205;
            if (y > 164 && y < 244) {
                int p = 12;
                v = (((x / p + y / p) % 2 == 0) ? 185 : 65);
            }
            if (std::abs(x - y) < 4) v = 240;
            v = std::max(0.0, std::min(v, 255.0));
            img.at<double>(y, x) = v;
        }
    }
    img.convertTo(img, CV_8U);
    return img;
}

// Reconstrução wavelet com threshold
static cv::Mat dwt_threshold_reconstruct(cv::Mat img, std::string wavelet="db4", int nivel=2,
double threshold=0.0) {
    cv::Mat img64f;
    img.convertTo(img64f, CV_64F);
    mm::WaveDec2 c = mm::wavedec2(img64f, wavelet, nivel);
    // Copia e aplica hard thresholding em todos os detalhes
    cv::Mat LL = c.LL.clone();
    std::vector<std::vector<cv::Mat>> coefs_t;
    coefs_t.push_back({LL});
    for (auto& detalhe : c.detail) {
        std::vector<cv::Mat> detalhe_t;
        for (auto& sb : detalhe) {
            detalhe_t.push_back(mm::wave_threshold(sb, threshold, "hard"));
        }
        coefs_t.push_back(detalhe_t);
    }
    // Reconstrução via waverec2
    mm::WaveDec2 rec_coefs;
    rec_coefs.LL = coefs_t[0][0];
    for (size_t j = 0; j < coefs_t.size()-1; j++) {
        rec_coefs.detail.push_back(coefs_t[j+1]);
    }
    cv::Mat rec = mm::waverec2(rec_coefs, wavelet);
    // Recorte para dimensão original
    rec = rec(cv::Rect(0, 0, img.cols, img.rows)).clone();
    cv::Mat rec8u;
    rec.convertTo(rec8u, CV_8U);
    return rec8u;
}

int main() {
```

```
    cv::Mat img_gray = gerar_imagem_sintetica(256);

    std::vector<int> thresholds = {0, 10, 30, 60, 100};
    std::vector<cv::Mat> imgs_thr;
    std::vector<std::string> titles_thr;

    imgs_thr.push_back(img_gray);
    titles_thr.push_back("Original");

    for (int t : thresholds) {
        cv::Mat rec = dwt_threshold_reconstruct(img_gray, "db4", 2, t);
        double psnr = cv::PSNR(img_gray, rec);
        imgs_thr.push_back(rec);
        std::string title = "T=" + std::to_string(t) + "  PSNR=" +
std::to_string(psnr).substr(0, std::to_string(psnr).find('.') + 2) + " dB";
        titles_thr.push_back(title);
    }

    // Converter vetor de cv::Mat para vetor de mm::Image
    std::vector<mm::Image> imgs_show;
    for (auto& img : imgs_thr) {
        imgs_show.push_back(mm::Image(img));
    }

    mm::show(imgs_show, MM_OUT, titles_thr, 3);

    return 0;
}
```

```
Overwriting tmp/fig_05_dwt_reconstrucao.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_dwt_reconstrucao.cpp -o
tmp/fig_05_dwt_reconstrucao -lopencv_stitching -lopencv_alphamat -lopencv_aruco
-lopencv_barcode -lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_dwt_reconstrucao \
  && test -f "tmp/fig_05_dwt_reconstrucao.png" \
  || echo "  mm::show não gravou tmp/fig_05_dwt_reconstrucao.png"
```

```
[1] Original
[2] T=0  PSNR=361.2 dB
[3] T=10  PSNR=42.4 dB
[4] T=30  PSNR=32.5 dB
[5] T=60  PSNR=28.0 dB
[6] T=100  PSNR=23.1 dB
```

```
try:
    mm.show(mm.read("tmp/fig_05_dwt_reconstrucao.png"), figsize=(14, 10))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_dwt_reconstrucao.png (ver a versao Python)")
```

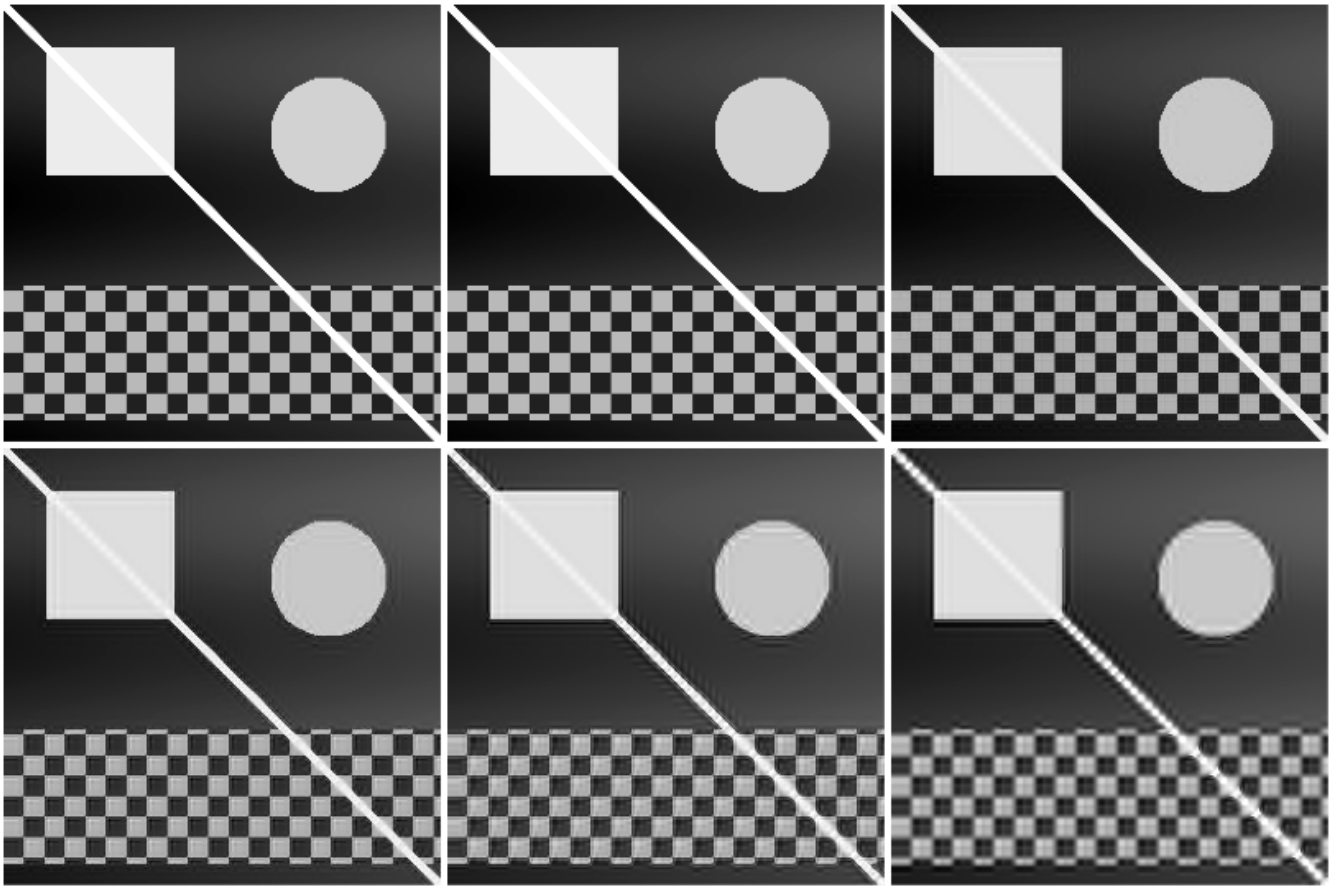


Figure 5.22: Reconstrução *wavelet* com limiarização de coeficientes (*hard thresholding*): à medida que o limiar aumenta, mais detalhes são zerados, produzindo imagens progressivamente mais suaves. Métrica PSNR quantifica a perda de qualidade sobre o padrão sintético.

Synthesis — Fourier vs. *Wavelets*: when to use each approach?

Table 5.5 summarizes the main structural and operational differences between the Discrete Fourier Transform (DFT) and the Discrete Wavelet Transform (DWT).

Table 5.5: Comparison between the Discrete Fourier Transform (DFT) and the Discrete Wavelet Transform (DWT), highlighting their main characteristics and applications.

Criterion	Fourier (DFT)	*Wavelet* (DWT)
Basis functions	Sinusoids with infinite support	Functions with compact support
Spatial localization	Not explicit (global)	Explicit (local)
Spectral filtering	Excellent for fine frequency control	Subband-based (scales)
Image compression	Basis of DCT (traditional JPEG)	Basis of DWT (JPEG 2000)
Multiscale analysis	No	Yes
Periodic noise removal	Highly efficient	Not recommended
Non-stationary signals	Limited	Highly efficient

In practical terms, the DFT stands out as the ideal tool for pure spectral analysis, design of selective filters in the frequency domain, and attenuation of periodic and harmonic noises. On the other hand, the DWT excels in scenarios that require rigorous preservation of the spatial localization of features associated with their frequency content, standing out in data compression, multiresolution analysis, and processing of abrupt

transitions. Thus, both transforms should be understood as perfectly complementary techniques, mapping distinct and specific paths for solving problems in DIP-CV.

i Analogies with Audio: Limitations and Cautions

When drawing analogies between image and audio processing, it is important to consider the fundamental differences:

- In stereo/multichannel audio systems, the phase between channels is crucial for the perception of spatial location (interaural phase and time differences).
- In monaural systems, phase has limited perceptual influence — the human ear is relatively insensitive to the absolute phase of isolated sinusoidal components.
- In images, the DFT phase is always fundamental for the spatial location of structures, regardless of whether it is a monochromatic or color image.

The analogy between phase in audio and phase in images should be used with caution, emphasizing that, although both carry information about the spatial/temporal organization of the signal, the perceptual mechanisms are fundamentally different.

5.7 Image Compression

While *wavelets* establish the theoretical foundation of the JPEG 2000 standard, the traditional JPEG standard is based on the **Discrete Cosine Transform (DCT)**. Despite structural differences, both approaches share the same fundamental principle: compacting image energy into a reduced number of coefficients and discarding the least relevant components with minimal visual impact.

The central goal of compression is to reduce the volume of data required for storing or transmitting an image. This process is made feasible by identifying and eliminating structural and perceptual **redundancies**.

5.7.1 Taxonomy of Redundancies

The development of compression algorithms is based on the identification and elimination of three main categories of redundancy, summarized in Table 5.6.

Table 5.6: Categories of redundancy in digital images and their respective exploration mechanisms.

Type	Definition	Exploration Approach
Spatial (interpixel)	High correlation and statistical dependence between neighboring pixels.	DCT, DWT, and predictive coding.
Spectral (interchannel)	Statistical correlation between the color channels of the same image.	Color space transformations (e.g., RGB to YC_bC_r).
Psychovisual	Insensitivity of the human visual system (HVS) to high-frequency and low-contrast variations.	Selective coefficient quantization processes.

Depending on the preservation of the original information after the decoding process, compression methods are divided into two fundamental classes:

- **Lossless:** Guarantees a bit-for-bit identical reconstruction of the original image. It is employed in scenarios where data integrity is strictly critical, such as in medical imaging, diagnostic imaging, and storage of textual documents.
- **Lossy:** Admits the introduction of controlled distortion in the signal in exchange for substantially higher compression rates. It is the standard approach for consumer photographs and video streaming, ecosystems in which the HVS tolerates small high-frequency attenuations without perceiving degradation of visual quality.

5.7.2 Discrete Cosine Transform (2D DCT-II)

The **Discrete Cosine Transform** (DCT) constitutes the central operation of the JPEG standard. Unlike the DFT, which uses a complex basis, the DCT is based on purely real trigonometric functions. For an image block $f(x, y)$ of dimensions $N \times N$, the **2D DCT-II** maps the spatial signal to the spatial frequency domain, generating the coefficient matrix $C(u, v)$ through:

$$C(u,v) = \alpha(u)\,\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y)\, \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right] \tag{5.9}$$

where the orthogonal normalization factors are given by $\alpha(0) = \sqrt{1/N}$ and $\alpha(k) = \sqrt{2/N}$ for $k > 0$.

Each coefficient $C(u, v)$ quantifies the contribution — or "weight" — of a specific spatial frequency within that block. The term $C(0, 0)$ is called the **DC component** and represents the average intensity of the block (zero frequency). The remaining coefficients, known as **AC components** (*Alternating Current*), correspond to progressively higher spatial frequencies.

5.7.3 The Basis Functions of the DCT

From a geometric perspective, Equation 5.9 performs the projection of the pixel block onto a set of orthogonal functions. For the standard JPEG case ($N = 8$), the spatial block is decomposed into a linear combination of **64 two-dimensional basis functions**, denoted by $B_{u,v}(x, y)$ and generated by the product of cosine functions:

$$B_{u,v}(x,y) = \cos\left[\frac{\pi(2x+1)u}{16}\right] \cos\left[\frac{\pi(2y+1)v}{16}\right]$$

Thus, the inverse operation can be interpreted as the exact reconstruction of the original block through the weighted sum of these 64 basis matrices, where each coefficient $C(u, v)$ acts as the analytical weight of its respective harmonic component.

The **spatial frequency** indicated by the indices (u, v) determines the number of oscillation cycles along the horizontal and vertical dimensions of the block. As illustrated in Figure 5.23 — whose code isolates each basis by applying the inverse transformation to unit impulses —, these 64 functions are organized into an 8×8 matrix. The upper-left corner $(u = 0, v = 0)$ displays the uniform zero-frequency (DC) pattern, while moving to the right (the u axis) or downward (the v axis) maps progressively larger harmonic variations, representing rapid transitions, edges, and textures in the horizontal, vertical, and diagonal orientations.

> **i** DCT vs. DFT: Advantage of Energy Compaction
>
> Both the DCT and the DFT map a spatial $N \times N$ block into a coefficient matrix of the same dimensions. However, for natural images, the DCT exhibits greater efficiency in **energy compaction** at low frequencies. This occurs because the DCT implicitly assumes an even symmetry of the signal at the block boundaries, which is equivalent to a continuous periodic extension, minimizing the effect of spectral spreading (*ringing*). As a result, most AC coefficients decay rapidly to values close to zero, optimizing the compression *pipeline* without introducing perceptible visual degradation.

```
%%writefile tmp/fig_05_dct_basis.cpp
#define MM_OUT "tmp/fig_05_dct_basis.png"
#include <opencv2/opencv.hpp>
#include <vector>
#include <string>


#include "morph.hpp"
```

```
#include <filesystem>

int main() {
    //| label: fig-05-dct-basis
    //| fig-cap: "The Visual Alphabet of JPEG: The 64 basis functions of DCT-II. The DC
    coefficient is at top-left (smooth). Going down and moving right, the spatial oscillation
    increases dramatically."
    //| echo: true
    //| output: true

    // Each basis is the IDCT of a single unit coefficient - assembled into an 8×8 mosaic.
    int tile = 32;
    cv::Mat montagem = cv::Mat::zeros(8 * tile, 8 * tile, CV_8UC1);
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            cv::Mat coef = cv::Mat::zeros(8, 8, CV_64F);
            coef.at<double>(i, j) = 1.0;
            cv::Mat base = mm::idct2(coef);
            cv::Mat base_norm;
            cv::normalize(base, base_norm, 0, 255, cv::NORM_MINMAX, CV_8U);
            cv::Mat base_resized;
            cv::resize(base_norm, base_resized, cv::Size(tile, tile), 0, 0,
cv::INTER_NEAREST);
            base_resized.copyTo(montagem(cv::Rect(j * tile, i * tile, tile, tile)));
        }
    }

    mm::Image montage_img = montagem;
    std::vector<mm::Image> imgs = {montage_img};
    std::vector<std::string> titles = {"The 64 DCT-II 8×8 bases (DC at top-left)"};
    mm::show(imgs, MM_OUT, titles, 1);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(montagem, "tmp/fig_05_dct_basis_0.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_05_dct_basis.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_dct_basis.cpp -o
tmp/fig_05_dct_basis -lopencv_stitching -lopencv_alphamat -lopencv_aruco -lopencv_barcode
-lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_dct_basis \
  && test -f "tmp/fig_05_dct_basis.png" \
  || echo "  mm::show não gravou tmp/fig_05_dct_basis.png"
```

```
[1] The 64 DCT-II 8×8 bases (DC at top-left)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_05_dct_basis_0.png"),
        ],
        titles=[
            'As 64 bases da DCT-II 8×8 (DC no topo-esquerdo)',
        ],
        cols=1,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_05_dct_basis_0.png
    (ver a versao Python)")
```

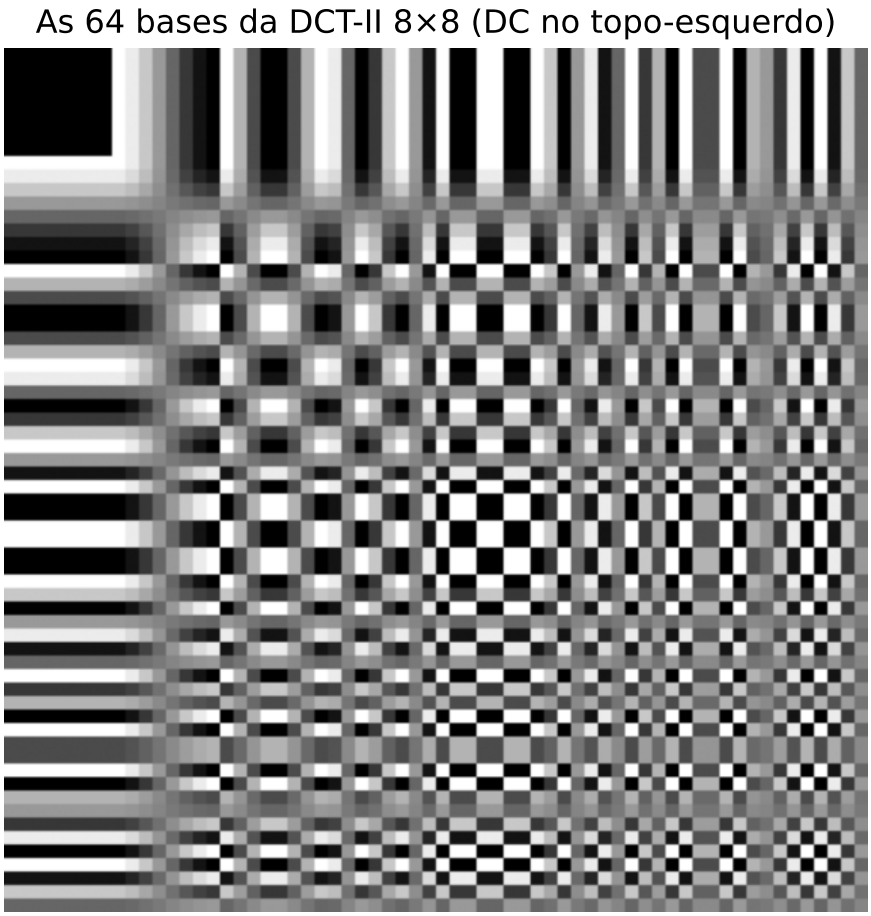


Figure 5.23: O Alfabeto Visual do JPEG: As 64 funções de base da DCT-II. O coeficiente DC fica no topo esquerdo (suave). Ao descer e avançar à direita, a oscilação espacial aumenta drasticamente.

5.7.4 Energy Concentration and Progressive Reconstruction

Before applying the DCT, the pixels of the intensity block are routinely shifted (by subtracting 128 for 8-bit images) in order to center the signal around zero, eliminating unnecessary DC components. When computing the DCT on the resulting block, the property of **energy compaction** becomes evident: almost all of the variance and information from the original image is concentrated in the DC coefficient ($C(0,0)$) and in the first low-frequency AC harmonics.

Figure 5.24 demonstrates this phenomenon through a progressive reconstruction by abrupt truncation. Instead of using all 64 coefficients, the algorithm preserves only the first k components—selected based on a scan that prioritizes low spatial frequencies—and nullifies the remaining ones.

The inverse synthesis (**IDCT**) performed with only a fraction of the coefficients (such as 15% or 30%) is already capable of recovering the structures and the macro illumination of the original pixel block. As higher-frequency harmonics are progressively reintroduced, fine details and rapid transitions are restored. This behavior validates the principle of perceptual compression: the discarded high frequencies carry little energy, and their absence, under normal conditions, has a secondary visual impact on the observer's perception.

```
%%writefile tmp/fig_05_dct_bloco.cpp
#define MM_OUT "tmp/fig_05_dct_bloco.png"
//| label: fig-05-dct-bloco
//| fig-cap: "DCT 2D em bloco 8×8: coeficientes e reconstrução progressiva."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <cmath>
#include <algorithm>
#include <cstdint>
#include "morph.hpp"

// DCT-II 2D ortogonal (separável)
cv::Mat dct2(const cv::Mat& bloco) {
    cv::Mat temp, result;
    cv::dct(bloco, temp);
    cv::dct(temp.t(), result);
    return result.t();
}

// IDCT-II 2D ortogonal
cv::Mat idct2(const cv::Mat& coefs) {
    cv::Mat temp, result;
    cv::idct(coefs, temp);
    cv::idct(temp.t(), result);
    return result.t();
}

int main() {
// [pdi:state-io] auto-generated - do not edit by hand
mm::Image img_gray = mm::_read_state("tmp/state/img_gray_20.png");
// [pdi:state-io:end]

    //   Bloco 8×8 centralizado da imagem
    int cy = img_gray.h / 2;
    int cx = img_gray.w / 2;

    // Extrair bloco 8×8 da imagem
    cv::Mat img_mat = img_gray;
    cv::Mat bloco_raw = img_mat(cv::Rect(cx, cy, 8, 8)).clone();
    bloco_raw.convertTo(bloco_raw, CV_64F);
    cv::Mat bloco = bloco_raw - 128.0;

    cv::Mat C = dct2(bloco);

    // Arredondar para impressão
    cv::Mat C_rounded;
    cv::Mat C_abs;
    cv::absdiff(C, cv::Scalar(0), C_abs);
    C_abs += 0.5;
    cv::Mat C_int;
```

```
    C_abs.convertTo(C_int, CV_32S, 1.0, 0);
    // Corrigir sinal
    cv::Mat C_sign = C.clone();
    cv::Mat C_int_sign;
    C_abs.convertTo(C_int_sign, CV_32S);
    for(int i = 0; i < C.rows; i++) {
        for(int j = 0; j < C.cols; j++) {
            if(C.at<double>(i,j) < 0) {
                C_int.at<int>(i,j) = -C_int.at<int>(i,j);
            }
        }
    }

    printf("Coeficientes DCT do bloco 8×8:\n");
    for(int i = 0; i < C_int.rows; i++) {
        for(int j = 0; j < C_int.cols; j++) {
            printf("%4d ", C_int.at<int>(i,j));
        }
        printf("\n");
    }

    double energia_dc = C.at<double>(0,0) * C.at<double>(0,0);
    double energia_total = 0.0;
    for(int i = 0; i < C.rows; i++) {
        for(int j = 0; j < C.cols; j++) {
            energia_total += C.at<double>(i,j) * C.at<double>(i,j);
        }
    }

    printf("\nEnergia DC     : %.1f\n", energia_dc);
    printf("Energia total  : %.1f\n", energia_total);
    printf("Fração no DC   : %.1f%% ← concentração de energia\n", 100.0 * energia_dc /
energia_total);

    //   Reconstrução progressiva
    std::vector<mm::Image> imgs_rec;
    std::vector<std::string> titles_rec;

    // Bloco original
    cv::Mat bloco_orig_disp;
    bloco_raw.convertTo(bloco_orig_disp, CV_8U);
    cv::Mat bloco_orig_norm = bloco_orig_disp + 128;
    imgs_rec.push_back(mm::Image(bloco_orig_norm));
    titles_rec.push_back("Bloco original\n(8×8 pixels)");

    std::vector<int> keeps = {1, 4, 10, 20, 40, 64};

    for(int keep : keeps) {
        cv::Mat C_trunc = cv::Mat::zeros(C.size(), C.type());

        // Gerar índices ordenados por soma u+v
        std::vector<std::pair<int,int>> indices;
        for(int u = 0; u < 8; u++) {
            for(int v = 0; v < 8; v++) {
                indices.push_back({u, v});
            }
        }
        std::sort(indices.begin(), indices.end(),
                  [](const std::pair<int,int>& a, const std::pair<int,int>& b) {
                      return (a.first + a.second) < (b.first + b.second);
                  });
```

```
        for(int i = 0; i < keep && i < (int)indices.size(); i++) {
            int u = indices[i].first;
            int v = indices[i].second;
            C_trunc.at<double>(u, v) = C.at<double>(u, v);
        }

        cv::Mat rec_temp = idct2(C_trunc) + 128.0;
        cv::Mat rec;
        cv::threshold(rec_temp, rec_temp, 255, 255, cv::THRESH_TRUNC);
        cv::threshold(rec_temp, rec_temp, 0, 0, cv::THRESH_TOZERO);
        rec_temp.convertTo(rec, CV_8U);

        imgs_rec.push_back(mm::Image(rec));

        double pct = (double)keep / 64.0 * 100.0;
        char title[100];
        snprintf(title, sizeof(title), "%d coef.\n(%.0f%% do total)", keep, pct);
        titles_rec.push_back(title);
    }

    mm::show(imgs_rec, MM_OUT, titles_rec, 4);

    return 0;
}
```

```
Overwriting tmp/fig_05_dct_bloco.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_dct_bloco.cpp -o
tmp/fig_05_dct_bloco -lopencv_stitching -lopencv_alphamat -lopencv_aruco -lopencv_barcode
-lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_dct_bloco \
  && test -f "tmp/fig_05_dct_bloco.png" \
  || echo "  mm::show não gravou tmp/fig_05_dct_bloco.png"
```

```
Coeficientes DCT do bloco 8×8:
  13   19   19   18   17   15   12    7
  21   28   25   22   21   18   15    9
  23   32   28   24   23   21   17    9
  24   34   34   37   40   36   25   12
  23   33   38   48   54   47   30   13
  22   32   37   46   53   48   32   13
  20   28   29   36   46   46   35   18
  13   17   17   22   30   34   28   16

Energia DC     : 168.5
Energia total  : 52234.0
Fração no DC   : 0.3% ← concentração de energia
[1] Bloco original
(8×8 pixels)
[2] 1 coef.
```

```
(2% do total)
[3] 4 coef.
(6% do total)
[4] 10 coef.
(16% do total)
[5] 20 coef.
(31% do total)
[6] 40 coef.
(62% do total)
[7] 64 coef.
(100% do total)
```

```
try:
    mm.show(mm.read("tmp/fig_05_dct_bloco.png"), figsize=(12, 7))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_05_dct_bloco.png
    (ver a versao Python)")
```

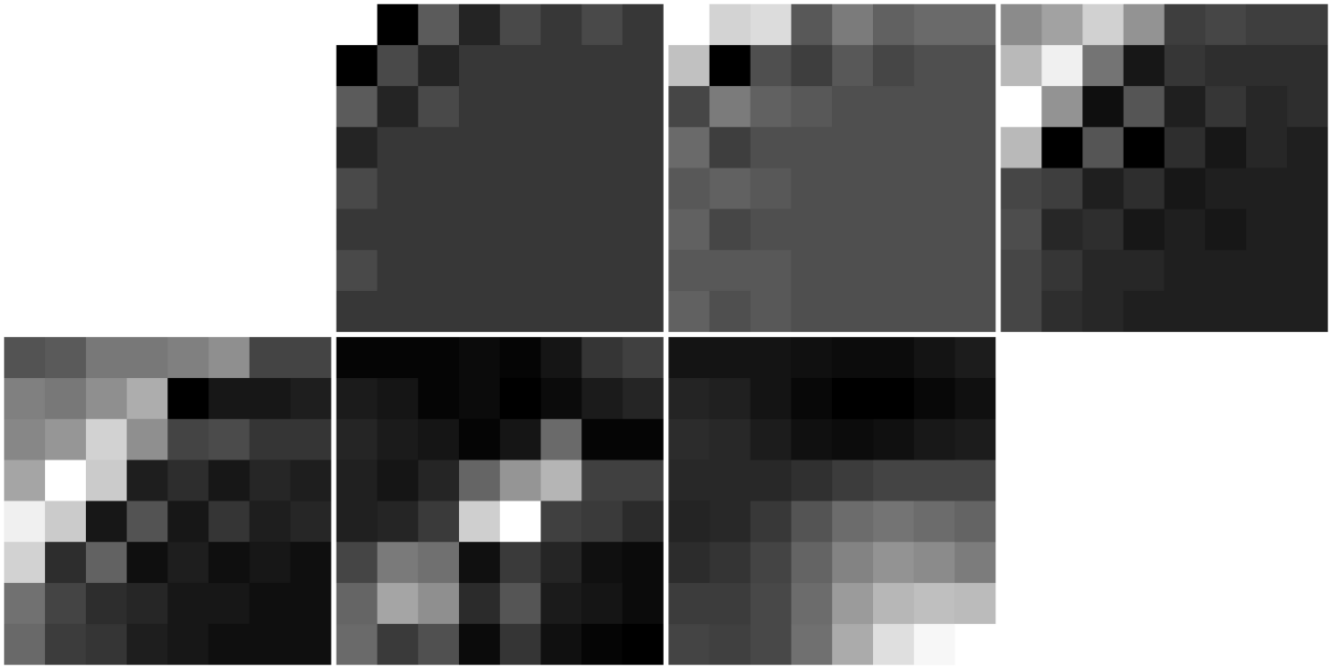


Figure 5.24: DCT 2D em bloco 8×8: coeficientes e reconstrução progressiva.

5.7.5 The JPEG Compression Pipeline

The JPEG standard operates by dividing the image into disjoint blocks of 8×8 pixels, processed through a sequence of spatial, perceptual, and statistical transformations. The complete encoding pipeline is structured into six main stages:

$$\text{RGB} \xrightarrow{\text{(1) } YC_bC_r} \xrightarrow{\text{(2) Subsampling}} \xrightarrow{\text{(3) Blocks } 8\times8} \xrightarrow{\text{(4) DCT}} \xrightarrow{\text{(5) Quantization}} \xrightarrow{\text{(6) Entropy Coding}}$$

Table 5.7 details the analytical function and the perceptual rationale that justifies each of these stages.

Table 5.7: Stages of the JPEG compression pipeline and their respective design rationales.

Stage	Operation	Perceptual and Statistical Rationale
1	Conversion $RGB \to YC_bC_r$	Separates luminance (Y) from chrominance (C_b, C_r). The human visual system (HVS) exhibits greater sensitivity to variations in brightness than to color.
2	Chrominance subsampling (e.g., 4:2:0)	Reduces the spatial resolution of the color channels by half, discarding redundant data with negligible visual impact.
3–4	Centering and application of the 8×8 DCT	Translates the pixels to the range $[-128, 127]$ and compacts the spectral energy of the block into the low-frequency coefficients.
5	Selective linear quantization	Divides each coefficient $C(u, v)$ by the corresponding element of the matrix $Q(u, v)$, applying integer rounding. This constitutes the primary source of lossy compression.
6	Zigzag scanning and coding	Orders the quantized coefficients to maximize consecutive null sequences, optimizing run-length encoding (RLE) and Huffman coding.

The **quantization matrix** $Q(u, v)$ is the central mechanism for controlling the trade-off between compression ratio and visual quality. In the practical algorithm of Figure 5.25, the quality factor specified by the user (on a scale from 1 to 100) is converted into a scalar that parameterizes the severity of the matrix Q. Reduced quality values expand the divisors of $Q(u, v)$, forcing the massive truncation of AC coefficients to zero. When this elimination is excessive, the discontinuity at the boundaries of adjacent blocks is not attenuated during reconstruction, giving rise to the so-called **blocking artifacts**.

The Logic of Zigzag Scanning

The efficiency of the entropy coder subsequent to quantization depends directly on the ordering of the data. Since the DCT concentrates the essential energy at the upper-left vertex of the matrix (low frequencies) and pushes the null coefficients toward the opposite ends, a linear reading by rows or columns would fragment the sequences of zeros.

Zigzag ordering solves this limitation by traversing the matrix diagonally in increasing order of spatial frequency. This mapping groups the significant coefficients at the beginning of the vector and concentrates the null coefficients into a single continuous sequence at the end of the arrangement, allowing the RLE algorithm to encode large blocks of data compactly and efficiently.

What is RLE?

RLE (*Run-Length Encoding*) is a lossless compression technique that encodes consecutive sequences of identical values — especially **zeros** — as a pair (count, value). In JPEG, after zigzag scanning, the quantized coefficients are organized so that the zeros are concentrated at the end of the vector. RLE then compresses this long run of zeros with extreme efficiency, optimizing the storage and transmission of the compressed image.

```
%%writefile tmp/fig_05_jpeg_pipeline.cpp
#define MM_OUT "tmp/fig_05_jpeg_pipeline.png"
//| label: fig-05-jpeg-pipeline
//| fig-cap: "*Pipeline* JPEG simplificado aplicado à imagem clássica do *Cameraman*: DCT em
blocos 8×8, quantização com diferentes fatores de qualidade e reconstrução via IDCT. Os
artefatos de bloco (*blocking artifacts*) tornam-se visualmente evidentes em fatores de
qualidade reduzidos ($Q=10$ e $Q=25$)."
//| echo: true
//| output: true

#include <iostream>
#include <vector>
#include <string>
#include <opencv2/opencv.hpp>
#include "morph.hpp"

int main() {
    // Pipeline JPEG (DCT 8x8 -> quantizacao -> IDCT) via mm.jpegCompress,
    // na imagem classica do Cameraman (asset do capitulo) reduzida a 256x256.
    cv::Mat img_src_cv = cv::imread("imagens/cameraman.png", cv::IMREAD_GRAYSCALE);
    cv::Mat img_resized;
    cv::resize(img_src_cv, img_resized, cv::Size(256, 256));
    mm::Image img_src = mm::gray(img_resized);

    std::vector<mm::Image> imgs = {img_src};
    std::vector<std::string> titles = {"Original (Cameraman)"};
    for (int q : {10, 25, 50, 75, 90}) {
        mm::Image rec = mm::jpegCompress(img_src, q);
        imgs.push_back(rec);
        double psnr_val = mm::psnr(img_src, rec);
        titles.push_back("Q=" + std::to_string(q) + " (PSNR=" +
                         std::to_string(psnr_val).substr(0, 4) + " dB)");
    }

    mm::show(imgs, MM_OUT, titles, 3);
    return 0;
}
```

```
Overwriting tmp/fig_05_jpeg_pipeline.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_jpeg_pipeline.cpp -o
tmp/fig_05_jpeg_pipeline -lopencv_stitching -lopencv_alphamat -lopencv_aruco -lopencv_barcode
-lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_jpeg_pipeline \
  && test -f "tmp/fig_05_jpeg_pipeline.png" \
  || echo "  mm::show não gravou tmp/fig_05_jpeg_pipeline.png"
```

```
[1] Original (Cameraman)
[2] Q=10 (PSNR=28.0 dB)
[3] Q=25 (PSNR=30.6 dB)
[4] Q=50 (PSNR=32.8 dB)
[5] Q=75 (PSNR=35.1 dB)
[6] Q=90 (PSNR=40.0 dB)
```

```
try:
    mm.show(mm.read("tmp/fig_05_jpeg_pipeline.png"), figsize=(14, 10))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_jpeg_pipeline.png (ver a versao Python)")
```

Figure 5.25: *Pipeline* JPEG simplificado aplicado à imagem clássica do *Cameraman*: DCT em blocos 8×8, quantização com diferentes fatores de qualidade e reconstrução via IDCT. Os artefatos de bloco (*blocking artifacts*) tornam-se visualmente evidentes em fatores de qualidade reduzidos ($Q = 10$ e $Q = 25$).

5.7.6 Interactive Simulator: DCT Quantization

The simulator in Figure 5.26 allows exploring the impact of the quantization process on an 8×8 block extracted from a real image, synthesizing in real time the following components:

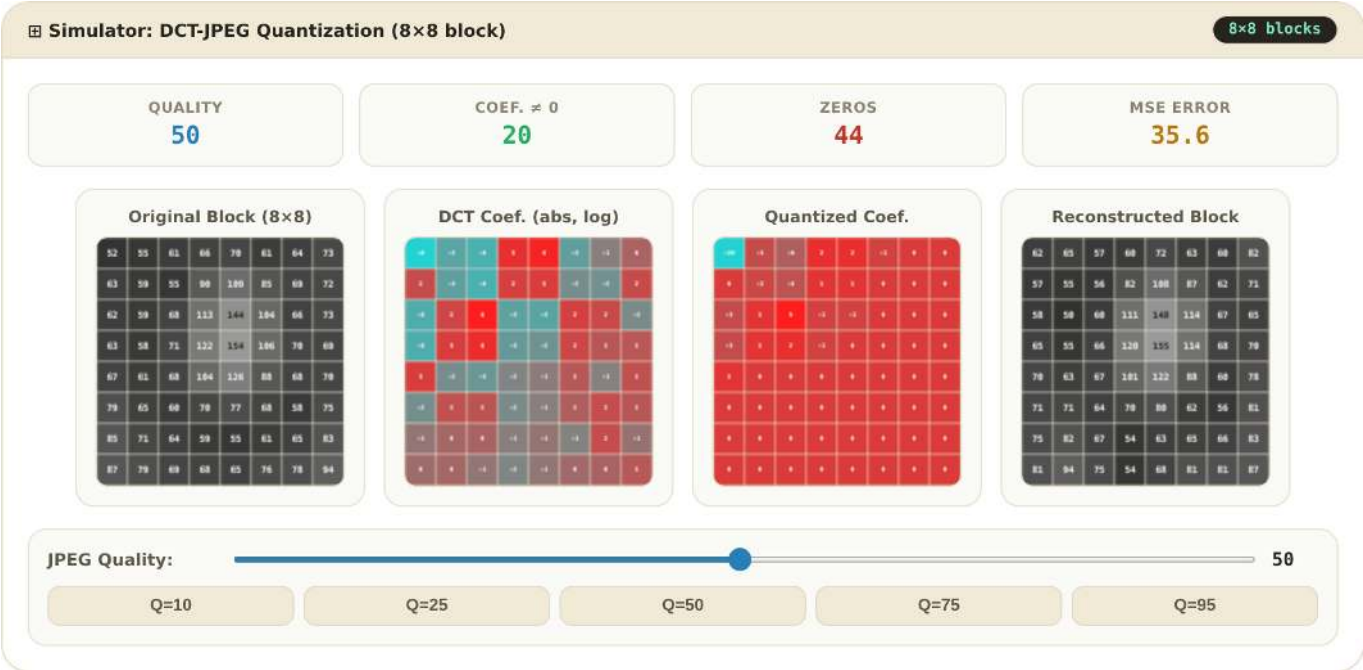
- **Original and reconstructed block:** Direct representation of pixels in the spatial domain in grayscale [0, 255].
- **DCT coefficients:** Energy distribution mapped logarithmically in a chromatic gradient, highlighting the concentration of intensity at the upper left vertex (low frequencies).
- **Quantized coefficients:** Display of integer values resulting from division by the matrix $Q(u, v)$, visually making explicit the mass emergence of null coefficients (in dark tones) as the quality factor is reduced.
- **Compression metrics:** Monitoring panel that quantifies the Mean Squared Error (MSE), the number of preserved coefficients, and the volume of zeros generated for entropy coding.

5.8 Comparison of Image Formats

The choice of a digital storage format directly impacts the trade-off between visual quality, file size, and computational decoding cost. The three most relevant formats for *web* architectures and visual computing systems are JPEG, PNG, and WebP.

5.8.1 Characteristics of the Formats

Table 5.8 synthesizes the structural properties of the main rasterized image formats.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap05/sim-05-dct.html

Figure 5.26: Interactive DCT-JPEG compression simulator: adjust the quality factor and visualize in real time the zeroed coefficients, the reconstructed block, and the quantization error.

Table 5.8: Structural comparison between the main rasterized image formats.

Characteristic	JPEG	PNG	WebP
Compression	Lossy	Lossless	Lossy and lossless.
Transparency (alpha channel)	No	Yes	Yes.
Animation support	No	Limited (APNG)	Yes.
Base algorithm	DCT + Huffman	DEFLATE (LZ77 + Huffman)	VP8 / VP8L.
Best for	Photography	Graphics, text, and icons	Universal use in a Web environment.
Worst for	Text and sharp edges	Complex photographic images	Legacy compatibility.

5.8.2 Quality Assessment Metrics

Two objective metrics are widely adopted to quantify the distortion introduced by compression processes:

Peak Signal-to-Noise Ratio (PSNR):

$$\text{PSNR} = 10 \log_{10}\left(\frac{L^2}{\text{MSE}}\right) \quad [\text{dB}] \tag{5.10}$$

where $L = 255$ for 8-bit quantized images and MSE represents the **Mean Squared Error**. PSNR values above 40 dB indicate excellent fidelity; between 30 dB and 40 dB represent good quality; and values below 30 dB correspond to easily perceptible visual degradations.

Structural Similarity Index (SSIM):

$$\text{SSIM}(f, g) = \frac{(2\mu_f\mu_g + c_1)(2\sigma_{fg} + c_2)}{(\mu_f^2 + \mu_g^2 + c_1)(\sigma_f^2 + \sigma_g^2 + c_2)} \tag{5.11}$$

SSIM evaluates local windows of the image based on three complementary components: **luminance** (μ_f, μ_g), **contrast** (σ_f, σ_g), and **structure** (σ_{fg}), weighted by stability constants c_1 and c_2. The index ranges over the interval $[-1, 1]$, where unity represents perfect identity. Unlike PSNR, SSIM considers the spatial organization of errors, aligning with the perception of the human visual system (HVS).

> **i** PSNR vs SSIM: Application of Perceptual Metrics
>
> PSNR has a simple mathematical formulation and low computational cost; however, it tends to overestimate quality in images with localized distortions or underestimate it in global brightness variations tolerated by the observer. SSIM models biological perception with greater fidelity but requires greater processing effort. For rigorous analyses of codecs, it is recommended to report both statistical metrics in a complementary manner.

5.8.3 Visual Inspection: Nature of Compression Artifacts

The mathematical nature of the codec dictates the type of degradation introduced at reduced bit rates. As illustrated in Figure 5.27, aggressive DCT-based compression in the JPEG standard segments the image into rigid grids, generating **blocking artifacts**. In contrast, algorithms based on predictive coding or representations subjected to advanced spatial transforms (such as WebP and JPEG 2000) eliminate block discontinuities but introduce loss of fine texture and characteristic blurring around high-contrast edges.

```
%%writefile tmp/fig_05_zoom_artefatos.cpp
#define MM_OUT "tmp/fig_05_zoom_artefatos.png"
//| label: fig-05-zoom-artefatos
//| fig-cap: "Análise comparativa de artefatos de compressão sob fator de qualidade reduzido
($Q=10$). À esquerda, observa-se o artefato de bloco característico da discretização por DCT
no JPEG. À direita, evidencia-se o efeito de atenuação e suavização de bordas intrínseco ao
padrão WebP."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <filesystem>
#include "morph.hpp"

cv::Mat zoom(const cv::Mat& img) {
    cv::Mat cropped = img(cv::Rect(150, 120, 80, 80));
    cv::Mat resized;
    cv::resize(cropped, resized, cv::Size(320, 320), 0, 0, cv::INTER_NEAREST);
    return resized;
}

int main() {
    // Read and resize the source image (grayscale conversion happens via mm::gray)
    cv::Mat img_original = mm::read("imagens/cameraman.png");
    cv::Mat img_resized;
    cv::resize(img_original, img_resized, cv::Size(256, 256), 0, 0, cv::INTER_LINEAR);
    mm::Image img_src_mm = mm::gray(mm::Image(img_resized));
    cv::Mat img_src = img_src_mm;  // Implicit conversion to cv::Mat (grayscale, 8-bit)

    // Save compressed versions with reduced quality
    std::filesystem::create_directories("tmp");
    cv::imwrite("tmp/zoom_q10.jpg",  img_src, {cv::IMWRITE_JPEG_QUALITY, 10});
    cv::imwrite("tmp/zoom_q10.webp", img_src, {cv::IMWRITE_WEBP_QUALITY, 10});

    // Load compressed versions (both saved as grayscale)
```

```
    cv::Mat z_orig = zoom(img_src);
    cv::Mat z_jpeg = zoom(cv::imread("tmp/zoom_q10.jpg",  cv::IMREAD_GRAYSCALE));
    cv::Mat z_webp = zoom(cv::imread("tmp/zoom_q10.webp", cv::IMREAD_GRAYSCALE));

    // Display results
    std::vector<std::string> titles = {
        "Zoom Original",
        "JPEG Q=10 (Artefato de Bloco)",
        "WebP Q=10 (Suavizacao)"
    };
    mm::show(std::vector<mm::Image>{z_orig, z_jpeg, z_webp}, MM_OUT, titles, 3);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(z_orig, "tmp/fig_05_zoom_artefatos_0.png");
mm::write(z_jpeg, "tmp/fig_05_zoom_artefatos_1.png");
mm::write(z_webp, "tmp/fig_05_zoom_artefatos_2.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_05_zoom_artefatos.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_zoom_artefatos.cpp -o
tmp/fig_05_zoom_artefatos -lopencv_stitching -lopencv_alphamat -lopencv_aruco -lopencv_barcode
-lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_zoom_artefatos \
  && test -f "tmp/fig_05_zoom_artefatos.png" \
  || echo "  mm::show não gravou tmp/fig_05_zoom_artefatos.png"
```

```
[1] Zoom Original
[2] JPEG Q=10 (Artefato de Bloco)
[3] WebP Q=10 (Suavizacao)
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_05_zoom_artefatos_0.png"),
            mm.read("tmp/fig_05_zoom_artefatos_1.png"),
            mm.read("tmp/fig_05_zoom_artefatos_2.png"),
        ],
        titles=[
            'Zoom Original',
            'JPEG Q=10 (Artefato de Bloco)',
            'WebP Q=10 (Suavizacao)',
        ],
        cols=3,
        figsize=(14, 5),
    )
```

```
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_zoom_artefatos_0.png (ver a versao Python)")
```

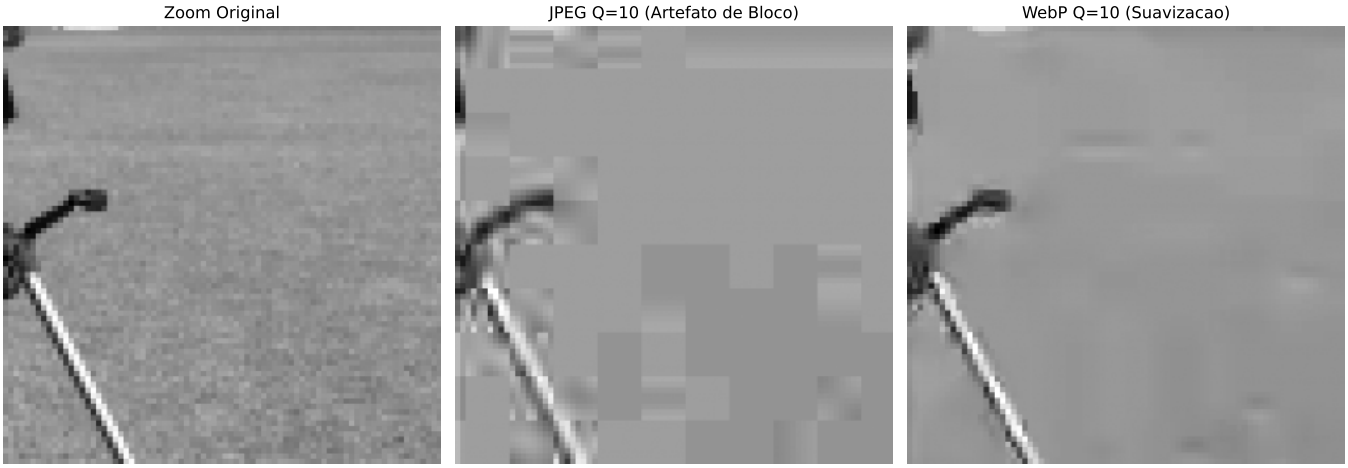


Figure 5.27: Análise comparativa de artefatos de compressão sob fator de qualidade reduzido ($Q = 10$). À esquerda, observa-se o artefato de bloco característico da discretização por DCT no JPEG. À direita, evidencia-se o efeito de atenuação e suavização de bordas intrínseco ao padrão WebP.

5.8.4 Quantitative and Spatial Assessment of Compression

Validation of lossy compression algorithms requires an analysis that correlates storage cost with the fidelity of the reconstructed signal. This assessment is carried out complementarily through global performance curves and the local mapping of distortions induced by the coders.

5.8.4.1 Rate-Distortion Curves

Figure 5.28 presents the empirical evaluation of the JPEG and WebP pipeline through **rate-distortion curves**, which monitor compression gain (file size in KB) as a function of PSNR. The PNG format serves as an ideal baseline (PSNR $= \infty$), as its *lossless* nature prevents any degradation, although it demands a substantially larger data volume.

Analysis of the curves demonstrates the superiority and efficiency of the WebP standard over traditional JPEG: to achieve the same level of mathematical fidelity (such as the excellent quality range, where PSNR > 40 dB), the WebP encoder generates significantly smaller files. This behavior reflects the practical impact of algorithmic evolution on optimizing digital transmission and storage systems.

```
%%writefile tmp/fig_05_formatos_comparacao.cpp
#define MM_OUT "tmp/fig_05_formatos_comparacao.png"
#include <opencv2/opencv.hpp>
#include <filesystem>
#include <string>
#include <vector>
#include "morph.hpp"


int main() {
    std::filesystem::create_directories("tmp");

    mm::Image src_img = mm::gray(mm::read("imagens/cameraman.png"));
    cv::Mat src = cv::Mat(src_img.h, src_img.w, CV_8UC1, src_img.data.data());
    cv::resize(src, src, cv::Size(256, 256));
```

```
    std::vector<double> jpeg_kb, jpeg_psnr;
    for (int q : {10, 20, 30, 40, 50, 60, 70, 80, 90, 95}) {
        std::string p = "tmp/fmt_q" + std::to_string(q) + ".jpg";
        cv::imwrite(p, src, {cv::IMWRITE_JPEG_QUALITY, q});
        cv::Mat rec = cv::imread(p, cv::IMREAD_GRAYSCALE);
        jpeg_kb.push_back((double)std::filesystem::file_size(p) / 1024.0);
        jpeg_psnr.push_back(mm::psnr(src, rec));
    }

    std::vector<double> webp_kb, webp_psnr;
    for (int q : {30, 50, 70, 85, 95}) {
        std::string p = "tmp/fmt_w" + std::to_string(q) + ".webp";
        cv::imwrite(p, src, {cv::IMWRITE_WEBP_QUALITY, q});
        cv::Mat rec = cv::imread(p, cv::IMREAD_GRAYSCALE);
        webp_kb.push_back((double)std::filesystem::file_size(p) / 1024.0);
        webp_psnr.push_back(mm::psnr(src, rec));
    }

    std::string p_png = "tmp/fmt.png";
    cv::imwrite(p_png, src, {cv::IMWRITE_PNG_COMPRESSION, 9});
    double png_kb = (double)std::filesystem::file_size(p_png) / 1024.0;

    std::vector<std::vector<double>> xs = {jpeg_kb, webp_kb};
    std::vector<std::vector<double>> ys = {jpeg_psnr, webp_psnr};
    std::vector<std::string> labels = {"JPEG", "WebP"};
    std::string title = "Curva Taxa-Distorcao (PNG sem perda: " +
                        std::to_string(png_kb).substr(0, std::to_string(png_kb).find('.') + 2)
+ " KB)";

    mm::Image chart = mm::lineChart(xs, ys, labels, {}, title,
                                    "Tamanho do arquivo (KB)", "PSNR (dB)");

    std::vector<mm::Image> imgs = {chart};
    std::vector<std::string> titles = {"JPEG x WebP x PNG"};
    mm::show(imgs, MM_OUT, titles, 1);


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(chart, "tmp/fig_05_formatos_comparacao_0.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_05_formatos_comparacao.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_formatos_comparacao.cpp
-o tmp/fig_05_formatos_comparacao -lopencv_stitching -lopencv_alphamat -lopencv_aruco
-lopencv_barcode -lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_formatos_comparacao \
  && test -f "tmp/fig_05_formatos_comparacao.png" \
```

```
  || echo "  mm::show não gravou tmp/fig_05_formatos_comparacao.png"
```

```
[1] JPEG x WebP x PNG
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_05_formatos_comparacao_0.png"),
        ],
        titles=[
            'JPEG x WebP x PNG',
        ],
        cols=1,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_formatos_comparacao_0.png (ver a versao Python)")
```

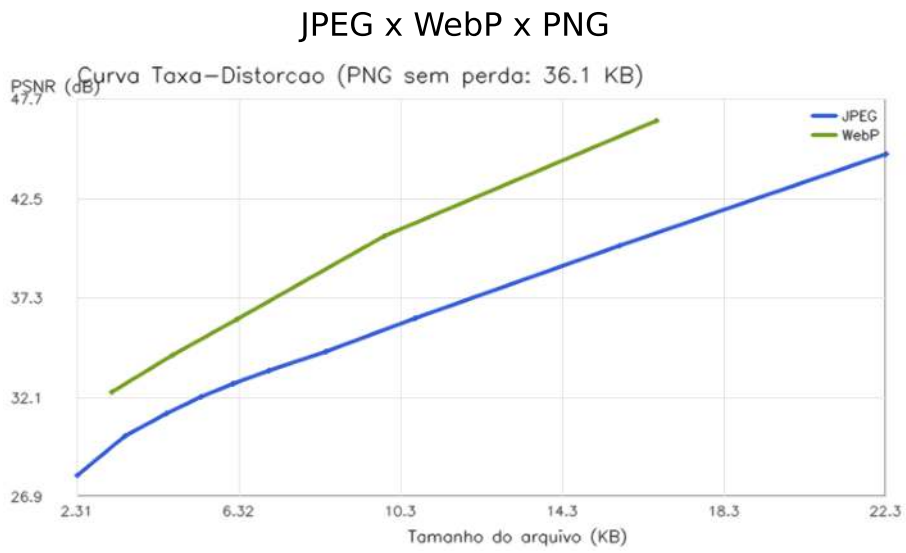


Figure 5.28: Curva taxa-distorção: PSNR vs tamanho de arquivo para JPEG, WebP e PNG aplicada à imagem do *Cameraman*.

i Original Image Size

The *Cameraman* image (256×256 pixels in grayscale) occupies **64 KB** in raw format (uncompressed). As a reference, *lossless* PNG compresses this volume to **36.2 KB** — highlighting that lossless compression already significantly reduces storage for images with homogeneous regions. In contrast, lossy formats (JPEG and WebP) achieve even smaller sizes: JPEG at quality 95 occupies 22.3 KB (PSNR 45 dB), while WebP at quality 90 reaches 12.5 KB with equivalent PSNR, demonstrating its superior compression efficiency.

5.8.4.2 Spatial Error Mapping and Perceptual Correlation

Although PSNR provides a rapid numerical indicator, global metrics fail to discriminate how information loss is geometrically distributed across the image. Figure 5.29 addresses this limitation by associating reconstructions at different qualities with their respective absolute error and SSIM maps.

The residual maps—obtained by the normalized absolute difference between the original and compressed images—reveal the intrinsic spatial signature of each coding architecture:

- **At high qualities ($Q = 95$ to $Q = 75$):** Distortions are predominantly concentrated around abrupt intensity transitions (edges), resulting from spectral mirroring caused by the discarding of high frequencies. The SSIM index remains close to unity, attesting to the integrity of the original structures.
- **At aggressive qualities ($Q = 50$ to $Q = 25$):** The error assumes a regularized orthogonal mesh structure. This geometric pattern highlights the emergence of **blocking artifacts**, indicating that severe quantization has corrupted the spatial correlation between adjacent 8×8 pixel blocks.

SSIM captures this morphological degradation far more sensitively than PSNR, penalizing the final score as structural organization and fine textures—to which the human visual system is highly responsive—are eliminated by the encoder.

```
%%writefile tmp/fig_05_ssim_artefatos.cpp
#define MM_OUT "tmp/fig_05_ssim_artefatos.png"
//| label: fig-05-ssim-artefatos
//| fig-cap: "Spatial degradation analysis: reconstructed images and respective normalized
absolute error maps for different JPEG quality factors."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <cmath>
#include <iostream>
#include "morph.hpp"

// SSIM implementation (Wang et al.) - standard windowed version
double compute_ssim(const cv::Mat& img1, const cv::Mat& img2) {
    const double C1 = 6.5025, C2 = 58.5225;
    cv::Mat I1, I2;
    img1.convertTo(I1, CV_32F);
    img2.convertTo(I2, CV_32F);

    cv::Mat I1_2 = I1.mul(I1);
    cv::Mat I2_2 = I2.mul(I2);
    cv::Mat I1_I2 = I1.mul(I2);

    cv::Mat mu1, mu2;
    cv::GaussianBlur(I1, mu1, cv::Size(11, 11), 1.5);
    cv::GaussianBlur(I2, mu2, cv::Size(11, 11), 1.5);

    cv::Mat mu1_2 = mu1.mul(mu1);
    cv::Mat mu2_2 = mu2.mul(mu2);
    cv::Mat mu1_mu2 = mu1.mul(mu2);

    cv::Mat sigma1_2, sigma2_2, sigma12;
    cv::GaussianBlur(I1_2, sigma1_2, cv::Size(11, 11), 1.5);
    sigma1_2 -= mu1_2;
    cv::GaussianBlur(I2_2, sigma2_2, cv::Size(11, 11), 1.5);
    sigma2_2 -= mu2_2;
    cv::GaussianBlur(I1_I2, sigma12, cv::Size(11, 11), 1.5);
    sigma12 -= mu1_mu2;

    cv::Mat t1 = 2 * mu1_mu2 + C1;
    cv::Mat t2 = 2 * sigma12 + C2;
    cv::Mat t3 = t1.mul(t2);

    cv::Mat t4 = mu1_2 + mu2_2 + C1;
    cv::Mat t5 = sigma1_2 + sigma2_2 + C2;
    cv::Mat t6 = t4.mul(t5);
```

```
    cv::Mat ssim_map;
    cv::divide(t3, t6, ssim_map);

    cv::Scalar mssim = cv::mean(ssim_map);
    return mssim[0];
}

int main() {
    // Cameraman (chapter asset), 256x256 - same image as the py track.
    cv::Mat img_full = cv::imread("imagens/cameraman.png", cv::IMREAD_GRAYSCALE);
    cv::Mat img_resized;
    cv::resize(img_full, img_resized, cv::Size(256, 256));
    mm::Image img_gray_mm = mm::gray(img_resized);
    cv::Mat img_gray(img_gray_mm.h, img_gray_mm.w, CV_8UC1, img_gray_mm.data.data());

    std::vector<mm::Image> imgs_ssim;
    std::vector<std::string> titles_ssim;
    imgs_ssim.push_back(img_gray_mm);
    titles_ssim.push_back("Original");

    for (int q : {25, 50, 75, 95}) {
        mm::Image rec_mm = mm::jpegCompress(img_gray_mm, q);  // DCT 8x8 -> quant -> IDCT
        cv::Mat rec = rec_mm;  // implicit conversion mm::Image -> cv::Mat
        double psnr_v = mm::psnr(img_gray_mm, rec_mm);

        cv::Mat img_gray_f, rec_f;
        img_gray.convertTo(img_gray_f, CV_32F);
        rec.convertTo(rec_f, CV_32F);

        cv::Mat diff = cv::abs(img_gray_f - rec_f);
        cv::Mat diff_vis;
        cv::normalize(diff, diff_vis, 0, 255, cv::NORM_MINMAX, CV_8U);

        double ssim_v = compute_ssim(img_gray, rec);

        char title1[100], title2[100];
        snprintf(title1, sizeof(title1), "Q=%d (PSNR=%.1fdB | SSIM=%.3f)", q, psnr_v, ssim_v);
        snprintf(title2, sizeof(title2), "Error map (Q=%d) - edges and blockiness", q);

        imgs_ssim.push_back(rec_mm);
        imgs_ssim.push_back(mm::Image(diff_vis));
        titles_ssim.push_back(title1);
        titles_ssim.push_back(title2);
    }

    mm::show(imgs_ssim, MM_OUT, titles_ssim, 3);
    return 0;
}
```

```
Overwriting tmp/fig_05_ssim_artefatos.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_ssim_artefatos.cpp -o
tmp/fig_05_ssim_artefatos -lopencv_stitching -lopencv_alphamat -lopencv_aruco -lopencv_barcode
-lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_ssim_artefatos \
  && test -f "tmp/fig_05_ssim_artefatos.png" \
  || echo "  mm::show não gravou tmp/fig_05_ssim_artefatos.png"
```

```
[1] Original
[2] Q=25 (PSNR=30.7dB | SSIM=0.860)
[3] Error map (Q=25) - edges and blockiness
[4] Q=50 (PSNR=32.8dB | SSIM=0.905)
[5] Error map (Q=50) - edges and blockiness
[6] Q=75 (PSNR=35.2dB | SSIM=0.938)
[7] Error map (Q=75) - edges and blockiness
[8] Q=95 (PSNR=44.8dB | SSIM=0.990)
[9] Error map (Q=95) - edges and blockiness
```

```
try:
    mm.show(mm.read("tmp/fig_05_ssim_artefatos.png"), figsize=(14, 14))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_ssim_artefatos.png (ver a versao Python)")
```

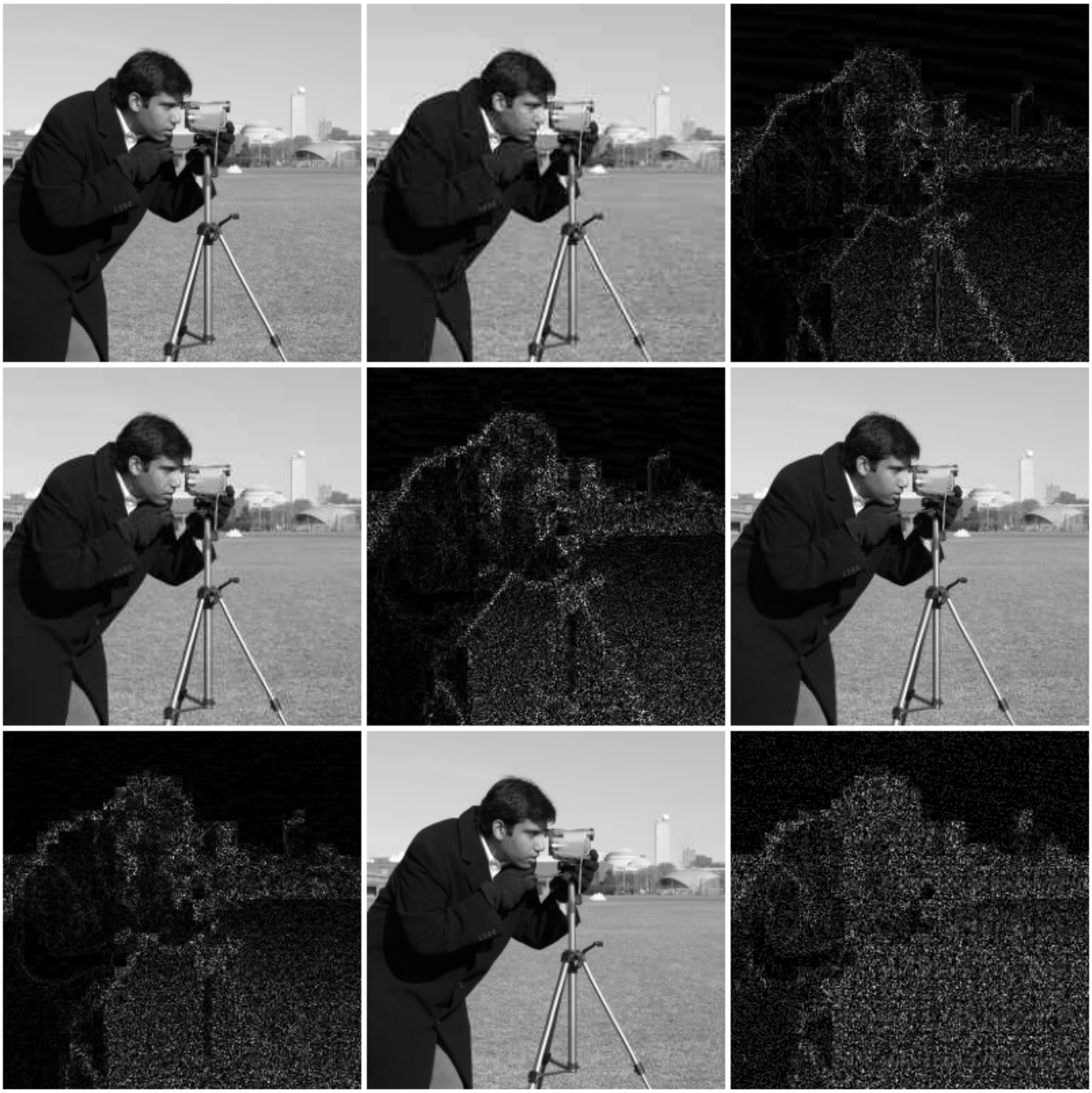


Figure 5.29: Análise espacial de degradação: imagens reconstruídas e respectivos mapas de erro absoluto normalizados para diferentes fatores de qualidade JPEG.

i Interpreting the Error Maps

The error maps presented were **individually normalized** (`cv2.NORM_MINMAX`) to maximize visual contrast and reveal the spatial structure of distortions. This means that:

- At **Q=95**, the absolute error is on the order of **0.5–1.5 gray levels** (visually imperceptible), but normalization amplifies it to black-and-white to highlight its location at edges and transitions.
- At **Q=25**, the absolute error is **10–20 times larger** (5–15 gray levels), but normalization also brings it to the same range [0, 255].

Therefore, **the intensity of white in the maps is NOT comparable across different qualities** — the maps serve only to reveal the **spatial signature** of the error (edges vs. blocks), not its magnitude. The correct magnitude is given by the PSNR and SSIM values, which clearly show that Q=95 has much

smaller error than Q=25.

Synthesis — JPEG Compression

The compression process in the JPEG standard is based on the combined application of spatial, perceptual, and statistical transformations to reduce the redundancies of an image. Table 5.9 summarizes the role of each stage in the *pipeline* and its respective impact on data reduction.

Table 5.9: Synthesis of the stages of the JPEG compression *pipeline* and their respective impacts.

Stage	Analytical Operation	Gain / Compression Mechanism
YC_bC_r **Conversion**	Isolation of luminance and chrominance channels.	Models the perception of the HVS, allowing color and brightness to be treated independently.
4:2:0 Subsampling	Reduction of the spatial resolution of the color channels (C_b and C_r).	Eliminates approximately 50% of raw data with minimal visual impact.
8×8 **DCT**	Mapping from the spatial domain to the spatial frequency domain.	Energy compaction, concentrating vital information in the first coefficients.
Linear Quantization	Integer division of the coefficients by a weighting matrix $Q(u, v)$.	Main source of lossy compression; eliminates imperceptible high frequencies.
Entropy Coding	Application of RLE algorithms and Huffman coding.	Lossless statistical compression, optimized by long runs of null coefficients.

Characteristic Degradation Artifacts

Applying excessively aggressive compression rates (reduced quality factors) introduces predictable distortions in the reconstructed image, arising from the mathematical limitations of the model:

- **Blocking artifacts:** Visible geometric discontinuities at the boundaries of 8×8 pixel blocks, caused by the loss of spatial correlation after severe quantization of the AC components.
- **Ringing effect:** Ghost oscillations or "smoke-like" distortions around sharp, high-contrast edges, caused by the abrupt elimination of high-frequency harmonics necessary to reconstruct step functions.
- **Loss of fine texture:** Attenuation of high-frequency, low-contrast details (such as grass, fabrics, or porosity), causing originally textured regions to assume an excessively smooth or homogenized appearance.

5.9 Practical Application: Noise Removal via Hybrid Filtering

Bringing together the techniques consolidated throughout this chapter, a complete **image restoration** *pipeline* is presented, combining spectral analysis in the frequency domain with adaptive filtering in the spatial domain. The goal is to attenuate mixed noise (composed of Gaussian degradation and periodic interference) while preserving the structural details of the original image as much as possible.

$$\text{Noisy Image} \xrightarrow{\text{FFT2}} \xrightarrow{\text{Gaussian Notch Filter}} \xrightarrow{\text{IFFT2}} \xrightarrow{\text{Bilateral Filter}} \text{Restored Image}$$

Complementary Evaluation: PSNR vs. SSIM

The pair of statistical metrics PSNR and SSIM provides a complementary qualitative and morphological evaluation of the restoration process:

- **PSNR:** Uniformly penalizes the pixel-by-pixel mean squared error.
- **SSIM:** Evaluates the preservation of perceptually relevant local structures (luminance, contrast, and edges).

In practice, there is an analytical trade-off between **noise reduction** and **detail preservation**: excessively aggressive spatial filters attenuate high-frequency noise well but degrade fine textures and smooth sharp edges — which **simultaneously reduces** both the PSNR and the SSIM relative to the original image. The challenge of filter design is to find the balance point that maximizes both metrics, ensuring a faithful and visually pleasing restoration.

5.9.1 Performance Analysis and Chapter Conclusion

The numerical and visual results generated by Figure 5.30 demonstrate the practical relevance of combining different processing domains. The simultaneous insertion of periodic and stochastic noise corrupts the morphological properties of the signal, severely reducing the similarity indices and the signal-to-noise ratio of the reference image.

The isolation and suppression of harmonic peaks in the frequency domain through the *notch* mask remove the sinusoidal interference fringes spread across the two-dimensional space. As evidenced in the printed data of Figure 5.30, this surgical filtering promotes an immediate and substantial leap in the PSNR metric. However, high-frequency Gaussian noise remains uniformly active in the spectrum, requiring a complementary approach.

The final restoration is consolidated in the spatial domain with the introduction of the bilateral filter. Unlike conventional low-pass operators (such as the Gaussian or mean filter), which would indiscriminately smooth noise and structural contours, bilateral filtering computes weights based on geometric proximity and radiometric intensity difference. This adaptive behavior attenuates remaining stochastic fluctuations in smooth transition regions while preserving the sharpness of spatial edges.

The convergence of both approaches results in a **substantial and simultaneous improvement** in PSNR and SSIM relative to the noisy image — although the final values remain inferior to those of the original image (PSNR $= \infty$, SSIM $= 1.0$), due to the inevitable loss of spectral and textural information during the filtering processes. The smooth (Gaussian) attenuation of spectrum peaks avoids *ringing* artifacts, while the bilateral filter eliminates residual stochastic noise without compromising edge sharpness. The results confirm the effectiveness and practical complementarity of the frequency analysis tools presented in this chapter, demonstrating that hybrid filtering (frequency + spatial) is superior to any isolated approach for restoring images degraded by mixed noise.

```
%%writefile tmp/fig_05_pipeline_denoising.cpp
#define MM_OUT "tmp/fig_05_pipeline_denoising.png"
//| label: fig-05-pipeline-denoising
//| fig-cap: "*Pipeline* completo de remoção de ruído misto: (1) adição de ruído gaussiano e
periódico; (2) identificação de picos de interferência no espectro de frequências; (3)
aplicação de máscara *notch* com atenuação gaussiana suave; (4) pós-processamento via filtro
bilateral para eliminação do ruído estocástico residual."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
#include <cmath>
#include <random>
#include <vector>
#include <string>
```

```cpp
#include <cstdio>
#include "morph.hpp"

// Helper function to suppress periodic peaks with a Gaussian notch
cv::Mat suprimir_pico_gaussiano(const cv::Mat& mask, int cy, int cx, double sigma = 3.0) {
    cv::Mat notch = cv::Mat::zeros(mask.size(), CV_64F);
    double sigma2 = 2.0 * sigma * sigma;

    for (int y = 0; y < mask.rows; y++) {
        double dy = y - cy;
        for (int x = 0; x < mask.cols; x++) {
            double dx = x - cx;
            double r2 = dy * dy + dx * dx;
            notch.at<double>(y, x) = std::exp(-r2 / sigma2);
        }
    }

    cv::Mat result;
    cv::subtract(cv::Mat::ones(mask.size(), CV_64F), notch, result);
    cv::multiply(mask, result, result);
    return result;
}

int main() {
    // ----- 1. Add mixed noise: Gaussian + periodic -----
    // Cameraman (chapter asset), 256x256 - same image as py track
    cv::Mat img_resized;
    cv::resize(cv::imread("imagens/cameraman.png", cv::IMREAD_GRAYSCALE), img_resized,
cv::Size(256, 256));
    mm::Image img_gray_mm = img_resized;
    cv::Mat img_gray = img_gray_mm;
    int h_img = img_gray.rows;
    int w_img = img_gray.cols;

    // ----- 2. Add noise -----
    std::mt19937 gen(42);
    std::normal_distribution<double> noise_dist(0.0, 15.0);

    // Gaussian noise + periodic noise
    cv::Mat img_float;
    img_gray.convertTo(img_float, CV_64F);

    cv::Mat ruido_gauss(h_img, w_img, CV_64F);
    cv::Mat ruido_period(h_img, w_img, CV_64F);

    int u0 = 15, v0 = 10;

    for (int y = 0; y < h_img; y++) {
        for (int x = 0; x < w_img; x++) {
            ruido_gauss.at<double>(y, x) = noise_dist(gen);
            // Periodic noise: 30 * sin(2  * (u0*x/w + v0*y/h))
            ruido_period.at<double>(y, x) = 30.0 * std::sin(2.0 * M_PI * (u0 * x /
static_cast<double>(w_img) +
                                            v0 * y / static_cast<double>(h_img)));
        }
    }

    cv::Mat img_noisy_float = img_float + ruido_gauss + ruido_period;
    cv::Mat img_noisy_8u;
    cv::threshold(img_noisy_float, img_noisy_float, 255.0, 255.0, cv::THRESH_TRUNC);
    cv::threshold(img_noisy_float, img_noisy_float, 0.0, 0.0, cv::THRESH_TOZERO);
```

```
    img_noisy_float.convertTo(img_noisy_8u, CV_8U);
    mm::Image img_noisy(img_noisy_8u);

    // ----- 3. Spectrum (log-magnitude) of the noisy image -----
    mm::Image mag_n = mm::spectrumMag(img_noisy);

    // ----- 4. Gaussian notch mask on the 4 periodic peaks -----
    cv::Mat mascara_notch = cv::Mat::ones(h_img, w_img, CV_64F);
    int cy0 = h_img / 2, cx0 = w_img / 2;

    int peak_params[4][2] = {{v0, u0}, {-v0, -u0}, {v0, -u0}, {-v0, u0}};
    for (auto& p : peak_params) {
        mascara_notch = suprimir_pico_gaussiano(mascara_notch, cy0 + p[0], cx0 + p[1], 3.0);
    }

    // ----- 5. Filtering: notch in frequency domain + bilateral -----
    mm::Image img_notch = mm::freqFilter(img_noisy, mascara_notch);

    // Convert to cv::Mat for bilateral filter
    cv::Mat img_notch_mat = img_notch;
    cv::Mat img_den_mat;
    cv::bilateralFilter(img_notch_mat, img_den_mat, 7, 25, 7);
    mm::Image img_den(img_den_mat);

    // Visualization of the mask
    cv::Mat mascara_vis_64f;
    cv::multiply(mascara_notch, cv::Scalar(255.0), mascara_vis_64f);
    cv::Mat mascara_vis;
    mascara_vis_64f.convertTo(mascara_vis, CV_8U);

    // PSNR calculations
    double psnr_n  = mm::psnr(img_gray_mm, img_noisy);
    double psnr_no = mm::psnr(img_gray_mm, img_notch);
    double psnr_d  = mm::psnr(img_gray_mm, img_den);

    printf("Ruidosa: PSNR=%.2f dB | Apos notch: %.2f dB | Notch+bilateral: %.2f dB\n",
           psnr_n, psnr_no, psnr_d);

    // ----- 6. Display results -----
    std::vector<mm::Image> images = {img_gray_mm, img_noisy, mag_n, mascara_vis, img_notch,
img_den};

    std::vector<std::string> titles = {
        "Original",
        "Ruidosa (PSNR=" + std::to_string(psnr_n).substr(0, 4) + " dB)",
        "Espectro (picos visiveis)",
        "Mascara notch (gaussiana)",
        "Apos notch (PSNR=" + std::to_string(psnr_no).substr(0, 4) + " dB)",
        "Notch + bilateral (PSNR=" + std::to_string(psnr_d).substr(0, 4) + " dB)"
    };

    mm::show(images, MM_OUT, titles, 6);

    return 0;
}
```

```
Overwriting tmp/fig_05_pipeline_denoising.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_pipeline_denoising.cpp
-o tmp/fig_05_pipeline_denoising -lopencv_stitching -lopencv_alphamat -lopencv_aruco
-lopencv_barcode -lopencv_bgsegm -lopencv_bioinspired -lopencv_ccalib -lopencv_dnn_objdetect
-lopencv_dnn_superres -lopencv_dpm -lopencv_face -lopencv_freetype -lopencv_fuzzy -lopencv_hdf
-lopencv_hfs -lopencv_img_hash -lopencv_intensity_transform -lopencv_line_descriptor
-lopencv_mcc -lopencv_quality -lopencv_rapid -lopencv_reg -lopencv_rgbd -lopencv_saliency
-lopencv_shape -lopencv_stereo -lopencv_structured_light -lopencv_phase_unwrapping
-lopencv_superres -lopencv_optflow -lopencv_surface_matching -lopencv_tracking
-lopencv_highgui -lopencv_datasets -lopencv_text -lopencv_plot -lopencv_ml -lopencv_videostab
-lopencv_videoio -lopencv_viz -lopencv_wechat_qrcode -lopencv_ximgproc -lopencv_video
-lopencv_xobjdetect -lopencv_objdetect -lopencv_calib3d -lopencv_imgcodecs -lopencv_features2d
-lopencv_dnn -lopencv_flann -lopencv_xphoto -lopencv_photo -lopencv_imgproc -lopencv_core \
  && ./tmp/fig_05_pipeline_denoising \
  && test -f "tmp/fig_05_pipeline_denoising.png" \
  || echo "  mm::show não gravou tmp/fig_05_pipeline_denoising.png"
```

```
Ruidosa: PSNR=20.20 dB | Apos notch: 22.36 dB | Notch+bilateral: 23.54 dB
[1] Original
[2] Ruidosa (PSNR=20.1 dB)
[3] Espectro (picos visiveis)
[4] Mascara notch (gaussiana)
[5] Apos notch (PSNR=22.3 dB)
[6] Notch + bilateral (PSNR=23.5 dB)
```

```
try:
    mm.show(mm.read("tmp/fig_05_pipeline_denoising.png"), figsize=(20, 4))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_pipeline_denoising.png (ver a versao Python)")
```

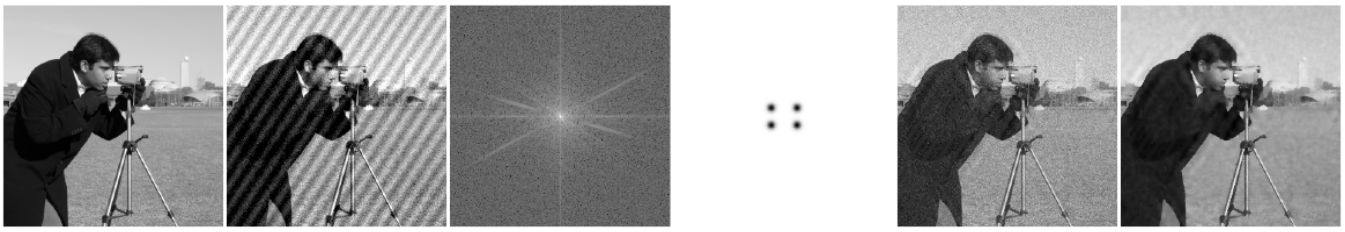


Figure 5.30: *Pipeline* completo de remoção de ruído misto: (1) adição de ruído gaussiano e periódico; (2) identificação de picos de interferência no espectro de frequências; (3) aplicação de máscara *notch* com atenuação gaussiana suave; (4) pós-processamento via filtro bilateral para eliminação do ruído estocástico residual.

5.10 Chapter Summary

The transition from the **spatial domain** to the **frequency domain** reveals the spectral energy distribution of the image, establishing the analytical foundation for advanced filtering, restoration, and data compression. The structural articulation of these concepts is synthesized in the conceptual map in Figure 5.31.

Essential Fundamentals

- **DFT and Visual Perception:** The spectrum decomposes the image into harmonic components. The **phase** retains the geometric intelligibility of the scene and the location of contours, while the **magnitude** dictates the distribution of contrast and global amplitudes.
- **Algorithmic Efficiency:** The Convolution Theorem enables the processing of large-scale masks in the frequency domain via FFT, reducing the asymptotic computational complexity from $O(N^2K^2)$ in space to $O(N^2 \log N)$.

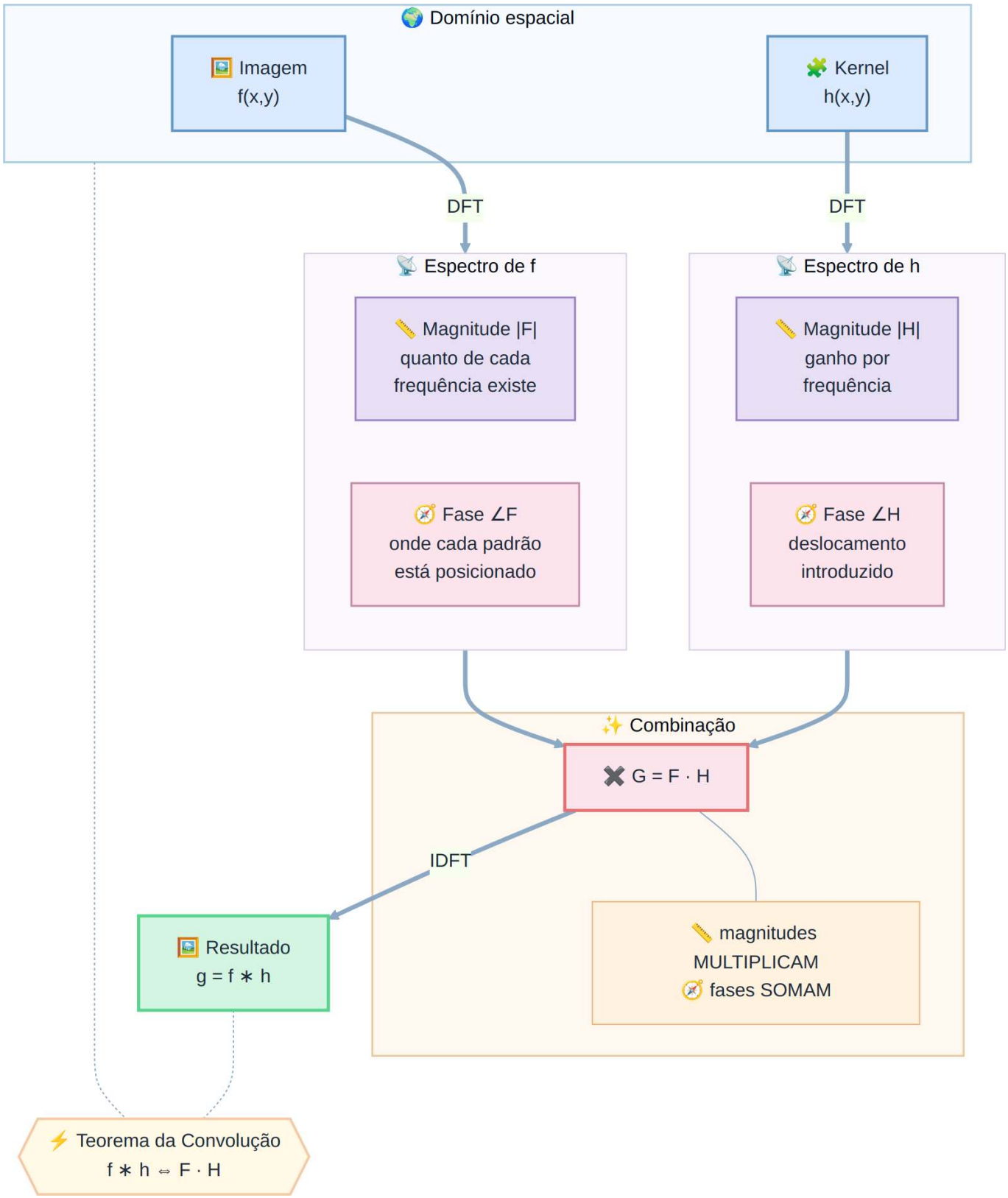


Figure 5.31: Conceptual map of transformations and properties in the frequency domain.

- **Ringing Phenomenon:** Abrupt cuts in the spectrum (Ideal Filters) generate unwanted spatial oscillations (Gibbs phenomenon). Smooth attenuation through **Butterworth** or **Gaussian** filters eliminates these discontinuities.
- **Multiresolution Analysis via Wavelets:** Overcoming the purely global nature of Fourier, the DWT captures frequency and spatial location simultaneously, underpinning the JPEG 2000 standard and supporting hierarchical representations analogous to feature extraction in Convolutional Neural Networks (CNNs).
- **Perceptual Compression (DCT):** The JPEG pipeline exploits the contrast limitations of the human visual system at high spatial frequencies. The DCT isolates the energy of 8×8 blocks, allowing quantization to discard AC coefficients of fine details without severe perceptual loss.

Next Steps: Chapter 6 inaugurates Part II of the work, applying image processing tools to solve real industrial inspection problems. Techniques for **segmentation and shape analysis** will be explored for the automatic detection of flaws in production lines—from identifying surface defects on parts to reading *QRCode* tags on exams, consolidating the bridge between the theory presented in Part I and the practical demands of computer vision.

5.11 🤖 Using Gemini Notebook as a Complementary Tutor

In this edition, the **Gemini Notebook** platform is encouraged as a complementary learning tool — **not as a substitute** for careful reading, problem-solving, or hands-on experimentation. Based on artificial intelligence architectures, the system uses exclusively the instructional material and documents provided by the author as its knowledge base, ensuring that the generated responses are conceptually aligned with the course content and the pedagogical approach adopted throughout this work.

Accessing the Intelligent Tutor

🚀 ACCESS Gemini Notebook: CHAPTER 05

🌐 Language and Programming Language

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

Guidelines on AI-Generated Content

Although artificial intelligence tools are efficient allies in the learning and review process, the generated content is subject to inconsistencies or technical inaccuracies. Therefore, systematic consultation of textbooks, scientific articles, and indexed academic sources is essential for the rigorous validation of information. The execution and modification of the practical Python examples provided in this chapter are strongly recommended as the primary method of experimental verification of the results.

5.12 Exercise List

1. **(10%) Direct Implementation of the 2D DFT:** Analytically implement the 2D Discrete Fourier Transform (DFT) without the aid of native library functions (such as `np.fft.fft2`), using strictly the mathematical formulation defined in Equation 5.1 for a matrix of dimensions 16×16. Perform numerical validation by comparing the generated coefficients with the results of the `np.fft.fft2` function, ensuring that the maximum absolute deviation is less than 10^{-8}. Measure the execution times of both methods and provide a theoretical justification for the observed disparity in terms of asymptotic complexity.

2. **(15%) Periodic Noise Suppression:** Add sinusoidal interferences with spatial frequencies $(u_0, v_0) \in$

$\{(5, 10), (20, 5), (30, 30)\}$ to the *Cameraman* test image. For each degradation scenario, design a specific notch filtering mask in the frequency domain to isolate and attenuate the unwanted harmonic peaks. Quantitatively evaluate the effectiveness of the restoration process by computing the PSNR and SSIM metrics. Discuss analytically the trade-off between sinusoidal noise attenuation and the undesirable attenuation of legitimate structural features of the image.

3. **(15%) Comparative Analysis of Low-Pass Operators:** Conduct a comparative study among the Ideal, Gaussian, and Butterworth low-pass filters (with harmonic orders $n = 1, 2, 4$), parameterized with cutoff frequencies $D_0 = 20, 40, 60$ pixels. For each structural combination, compute the PSNR and SSIM indices of the resulting image against the original reference signal. Organize the quantitative data in a structured table and plot the one-dimensional graphs of the corresponding transfer functions along the horizontal profile $H(u, 0)$.

4. **(15%) Haar Multiresolution Filter Bank:** Develop a script to manually perform the first-level 2D discrete wavelet decomposition using the Haar family. The algorithm must compute the corresponding low-pass (h) and high-pass (g) filter coefficients, applying them separably over the rows and columns of the matrix, followed by the decimation operation (spatial subsampling by a factor of 2). Numerically validate the accuracy of your implementation by comparing the obtained subbands with the output of the `pywt.dwt2(img, 'haar')` function.

5. **(15%) Sparse Compression by Wavelet Thresholding:** Apply the hard thresholding filtering technique to the detail coefficients of the wavelet decomposition, adopting the numerical thresholds $T \in \{5, 10, 20, 40, 80\}$ for the Haar, Daubechies (`db4`), and Symlets (`sym4`) families. After performing the synthesis process through the inverse transform (`pywt.waverec2`), compute the PSNR and SSIM values of each reconstructed image. Identify and justify which combination of wavelet family and threshold T maximizes structural similarity.

6. **(15%) Construction of a Simplified JPEG Encoder:** Implement the complete data compression pipeline simulating the JPEG standard. The workflow must encompass: spatial conversion $RGB \rightarrow YC_bC_r$, chroma subsampling at a 4:2:0 ratio, segmentation of luminance into disjoint blocks of 8×8 pixels, application of the orthogonal 2D DCT-II, and linear quantization based on the normalized luminance matrix scaled by desired quality factors. Perform the inverse decoding and quantitatively compare the reconstructions with the files generated by the `cv2.imencode` function for quality factors of 20, 50, and 80.

7. **(15%) Perceptual Analysis on Heterogeneous Content:** Develop a synthetic image composed of three distinct regions with contrasting spectral characteristics: a complex photographic texture (representing high stochastic frequencies), a vectorized text area with sharp edges (representing pure step transitions), and a continuous linear gradient (representing homogeneous low frequencies). Submit this mixed image to compression processes under the JPEG, PNG, and WebP formats. Evaluate and interpret the results by correlating the final file size on disk with the obtained PSNR and SSIM metrics, justifying which format exhibits the best performance for signals of a heterogeneous nature and why this advantage occurs in terms of energy compaction and perceptual preservation.

Chapter References

The theoretical foundation and analytical development of the concepts addressed in this chapter are based on the following reference works:

- **Gonzalez (2018)** — Classical formulations of 2D Discrete Fourier Transforms (DFT), design of analytical filters in the frequency domain, Discrete Cosine Transform (DCT), and fundamental principles of image compression systems.
- **Oppenheim (2010)** — Formal theory of signals and systems applied in the discrete domain, covering the mathematical properties of the DFT and the analytical modeling of the Convolution Theorem.
- **Mallat (1999)** — Mathematical foundation of wavelet theory, formalization of multiresolution analysis (MRA), and architecture of dyadic filter banks.

- **Wallace (1991)** — Original specification and engineering aspects of the ISO/IEC JPEG compression standard, with emphasis on psychovisual criteria for the design of DCT quantization matrices.
- **Szeliski (2022)** — Computational modeling and characterization of modern fidelity and perceptual quality metrics (PSNR and SSIM), as well as the comparative analysis of high-performance rasterized image formats.

5.13 💻 Practical Part with Programming Exercises

The present list of programming exercises (EP) consolidates the theoretical formulations presented throughout Chapter 5 — Transforms and Compression — through an applied practical track. The exercises are structured around matrices of reduced dimensions, enabling analytical validation and manual inspection of each coefficient, while maintaining the methodological consistency adopted in previous chapters.

The sequencing of the exercises strictly reproduces the conceptual flow of the chapter: it begins with the explicit implementation of the Discrete Fourier Transform (DFT) from its fundamental mathematical definition; it proceeds to the design of low-pass filters and *notch* masks in the frequency domain; it applies coefficient quantization (the core of lossy compression); and it concludes with the integration of these stages in the construction of a simplified JPEG compression *pipeline* and in the perceptual analysis of image formats.

> ❗ Guidelines for Solving the Programming Exercises
>
> In all exercises in this chapter, the coordinates of the **spectrum center** (the origin of spatial frequencies after the application of the `fftshift` shift) must be determined using integer division. For a matrix with L rows and C columns, the zero-frequency component is located at the position:
>
> $$(c_y, c_x) = \left(\left\lfloor \frac{L}{2} \right\rfloor, \left\lfloor \frac{C}{2} \right\rfloor \right)$$
>
> This convention is strictly identical to that adopted by the `np.fft.fftshift` function. Furthermore, in all stages that require discretization or numerical rounding (whether in the quantization of AC coefficients or in the final reconstruction of pixels), the standard rounding to the nearest integer (*round half away from zero*) must be employed, mitigating ambiguities in values with a fraction exactly equal to 0.5.

🎯 Objective of this Notebook

The notebook enables the development, validation, organization, and testing of solutions for **Programming Exercises (PEs)** in interactive environments, such as Colab, using the same test cases as Moodle, copying them there only when recording the official grade.

Download

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

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)

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, cpp=True)
from morph import mm
from testsuite import TestSuite
```

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

Running the Tests

To evaluate the tests, run `TestSuite("EP05_01.extensão").run()` in a new cell, replacing the extension with the one used for your language (`.py`, `.java`, `.c`, `.cpp`, `.js`, or `.r`). The system downloads the test cases from GitHub, runs the program, and calculates the grade automatically.

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

```
codigo = """
from morph import mm
# ... your code here ...
"""
TestSuite("EP05_01").run_code(codigo)
```

5.13.1 EP05_01 🟢 Ideal Low-Pass Filter by Distance in the Spectrum

In an **old document scanner**, the sensor captures crumpled paper and fiber texture along with the text — high-frequency noise that "pollutes" the spectrum at the edges. The maintenance technician has no access to the original image, only to the **magnitude spectrum already computed** by the scanner's software. Their job is simple and surgical: keep only the **central circle** of low frequencies (the global structure of the document) and erase everything outside the radius D_0, eliminating the fine texture without even needing to touch the spatial image.

This is the **Ideal Low-Pass Filter (LPFI)**: the most direct spectral operation in the chapter, but also the one that best reveals the anatomy of a centered spectrum.

5.13.1.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns) from the magnitude spectrum — already provided **centered** (equivalent to the output of `np.fft.fftshift`).
2. **Cutoff frequency:** Read the integer D_0.
3. **Data:** Read the integer values of the magnitude matrix, row by row.
4. **Spectrum center:** Compute $(c_y, c_x) = (L \,//\, 2,\ C \,//\, 2)$.
5. **Distance:** For each position (u, v), compute

$$D(u,v) = \sqrt{(u - c_y)^2 + (v - c_x)^2}$$

6. **Ideal mask:** Apply

$$H(u,v) = \begin{cases} 1, & D(u,v) \le D_0 \\ 0, & D(u,v) > D_0 \end{cases}$$

7. **Filtering:** The output value is $\text{mag}'(u,v) = \text{mag}(u,v) \cdot H(u,v)$.
8. **Output:** Display the filtered matrix with dimensions $L \times C$.

5.13.1.2 📌 Computational Constraints

- **Non-strict comparison:** the criterion uses $D(u,v) \le D_0$ (the boundary belongs to the filter, i.e., it is kept).
- **Type:** all input and output values are integers; the distance is computed in floating point only internally.

- **No magnitude rounding:** since the input is already integer and the mask is binary (0 or 1), the output never requires rounding.

5.13.1.3 🧠 Theoretical Background

Region	Distance to center	Filter effect
Center ($D \leq D_0$)	Low frequencies	Preserved — global structure maintained
Edges ($D > D_0$)	High frequencies	Zeroed — texture and noise removed
Small D_0	—	Reconstructed image would be very blurry
Large D_0	—	Little filtering; almost all energy preserved

5.13.1.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer D_0.
- Following lines: Integer elements of the magnitude matrix (centered).

Output:

- Filtered matrix with L rows and C columns, separated by spaces.

5.13.1.5 📌 Examples

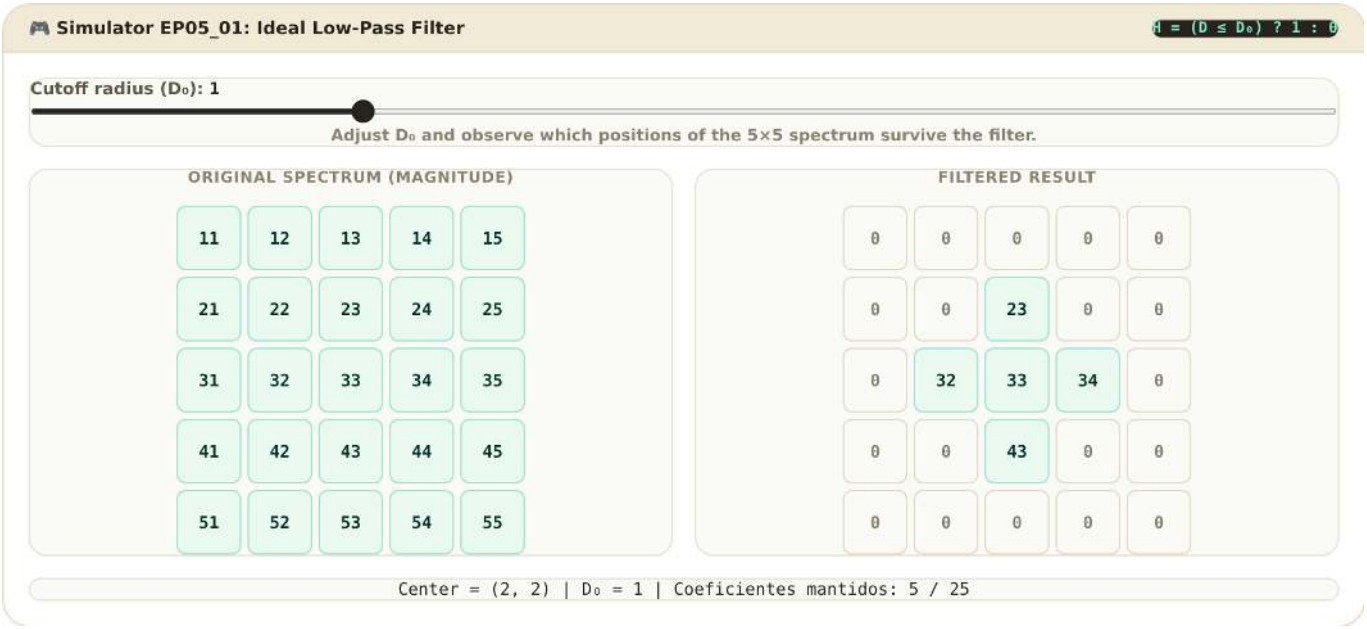
Input	Output	Observation
3 3 1 10 20 30 40 50 60 70 80 90	0 20 0 40 50 60 0 80 0	Center $(1, 1)$. Corners have $D = \sqrt{2} \approx 1.41 > 1$, hence they are zeroed; orthogonal neighbors have $D = 1 \leq 1$ and are kept.
1 3 0 5 9 7	0 9 0	$L = 1, C = 3$: center at $(0, 1)$. Only the central position itself ($D = 0$) survives $D_0 = 0$.

```
%%writefile EP05_01.cpp
// your solution
```

```
Overwriting EP05_01.cpp
```

```
TestSuite("EP05_01.cpp").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap05/sim-ep0501.html

Figure 5.32: EP05_01 Simulator: Ideal Low-Pass Filter in the Spectrum

5.13.2 EP05_02 🟡 *Notch* Filter: Removing Periodic Peaks

An **industrial inspection** camera captures images of circuit boards, but the production line's power supply introduces a **periodic electrical interference** — a stripe pattern almost imperceptible to the naked eye, yet visible in the Fourier spectrum as **pairs of bright peaks** symmetrically positioned around the center. The computer vision team cannot redo the capture: they must **surgically locate and erase** these peak pairs in the spectrum, preserving all other useful image information.

This is the role of the **notch reject filter**: unlike a low-pass filter (which affects a continuous region), it targets **specific points and their symmetric counterparts**, leaving the rest of the spectrum untouched.

5.13.2.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns) of the centered magnitude spectrum.
2. **Data:** Read the integer values of the magnitude matrix, row by row.
3. **Peaks:** Read the integer K (number of peak pairs to remove).
4. **For each of the K peaks:** read three integers Δv, Δu, r — vertical offset, horizontal offset, and notch radius.
5. **Spectrum center:** $(c_y, c_x) = (L \,//\, 2,\ C \,//\, 2)$.
6. **Symmetric suppression:** for each peak, zero out **all** positions (u, v) such that the distance to the point $(c_y + \Delta v,\ c_x + \Delta u)$ is $\leq r$, **and also** all positions with distance $\leq r$ to the symmetric point $(c_y - \Delta v,\ c_x - \Delta u)$.
7. **Output:** Display the resulting matrix with dimensions $L \times C$.

5.13.2.2 📌 Computational Constraints

- **Mandatory symmetry:** each reported peak generates **two** zeroed disks (the point and its symmetric counterpart relative to the center) — forgetting the symmetric point is the most common mistake.
- **Overlap:** if two disks overlap, the position remains zeroed (there is no "addition" or restoration).
- **Non-strict comparison:** a position is zeroed if distance $\leq r$.
- **Reading order:** the K peaks must be processed in the order they appear in the input, but the final result is independent of order (zeroing operations are commutative).

5.13.2.3 🧠 Theoretical Foundation

Concept	Role in the notch filter
Peak at $(\Delta v, \Delta u)$	Frequency of the periodic interference visually detected in the spectrum
Symmetric point $(-\Delta v, -\Delta u)$	Every DFT of a real signal is Hermitian: peaks always appear in pairs symmetric about the center
Radius r	Controls the "width" of rejection — a large r removes more energy around the peak, but also useful information

5.13.2.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Following lines: Integer elements of the magnitude matrix (centered), L lines.
- Next line: Integer K.
- Following K lines: three integers Δv, Δu, r (space-separated).

Output:

- The resulting matrix in L lines and C columns, space-separated.

5.13.2.5 📌 Examples

Input	Output	Observation
5 5 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 1 1 1 0	1 2 3 4 5 6 0 8 9 10 11 12 13 14 15 16 17 18 0 20 21 22 23 24 25	Center $(c_y, c_x) = (2, 2)$. Reported peak $(\Delta v, \Delta u) = (1, 1)$ generates the point $(3, 3)$ (value 19) and its symmetric counterpart $(1, 1)$ (value 7), both zeroed with $r = 0$ (only the exact points).

```
%%writefile EP05_02.cpp
// your solution
```

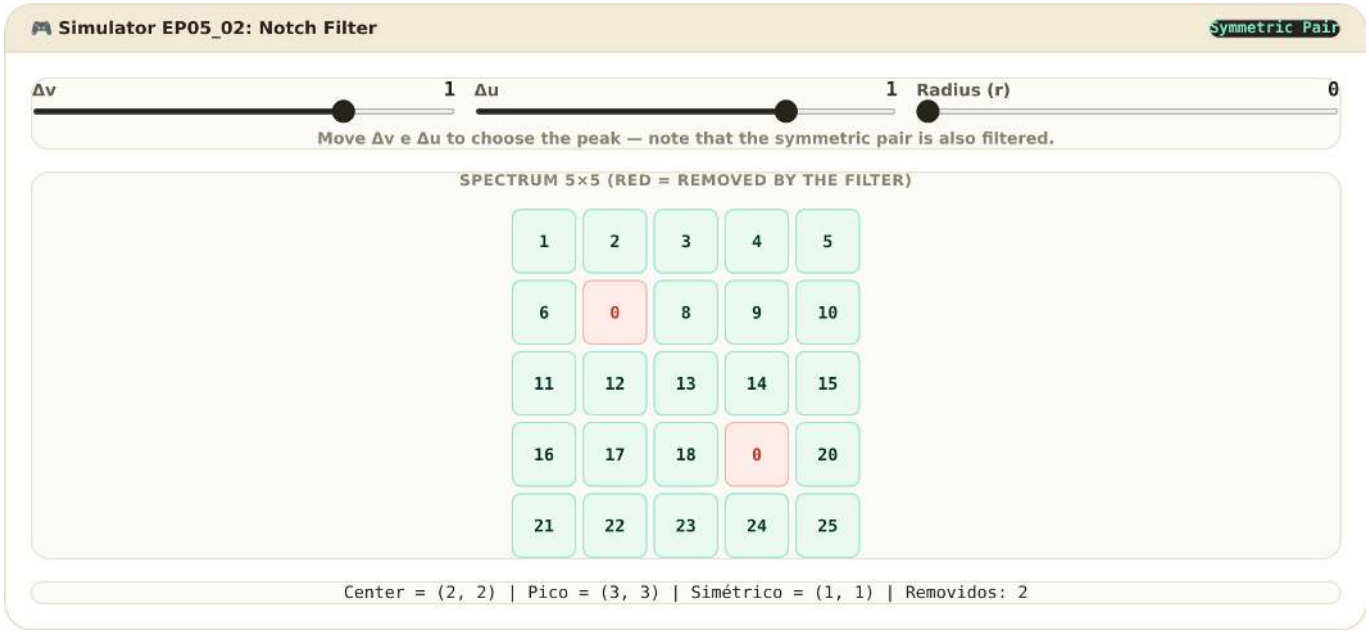
```
Overwriting EP05_02.cpp
```

```
TestSuite("EP05_02.cpp").run()
```

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

5.13.3 EP05_03 🟠 DCT Quantization: The True Source of Compression

A **photo gallery** application needs to reduce the size of thousands of images before *uploading* them to the cloud, without recoding everything from scratch. The engineer in charge already has the **DCT coefficients** for each 4×4 block calculated (the computationally expensive step has already been done) — only the **quantization table** needs to be applied, the step that actually discards information and generates compression. High-frequency coefficients, which are less perceptible to the human eye, receive large divisors



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap05/sim-ep0502.html

Figure 5.33: EP05_02 Simulator: Notch Filter

and tend to become **zero**; low-frequency coefficients, which are more perceptible, receive small divisors and survive almost intact.

You will implement exactly this step: **quantize and dequantize** (divide, round, multiply back) — the heart of JPEG *lossy* compression.

5.13.3.1 📋 Implementation Guidelines

1. **Block size:** Read the integer N ($N \times N$ block).
2. **Coefficients:** Read the matrix C of DCT coefficients, N rows with N integers each (they may be negative).
3. **Quantization table:** Read the matrix Q, N rows with N positive integers each.
4. **Quantization:** For each position (u, v), compute the quantized index

$$\tilde{C}(u,v) = \text{round}\left(\frac{C(u,v)}{Q(u,v)}\right)$$

   using standard rounding to the nearest integer (intermediate `.5` values never occur in the test cases).
5. **Dequantization (reconstruction):** Compute

$$C'(u,v) = \tilde{C}(u,v) \times Q(u,v)$$

6. **Output:** Display the reconstructed matrix C', $N \times N$, integers.

5.13.3.2 📌 Computational Constraints

- **Complete *round-trip*:** The output is the **reconstructed** coefficient ($\tilde{C} \times Q$), not the isolated quantized index.
- **Floating-point division:** The division $C(u,v)/Q(u,v)$ must be performed in floating point before rounding — truncated integer division will produce an incorrect result.
- **Preserved sign:** Negative coefficients retain their sign after quantization and reconstruction.
- $Q(u,v) > 0$ **always:** There is no need to handle division by zero.

5.13.3.3 🧠 Theoretical Background

Coefficient	Frequency	Typical value of Q	Effect of quantization
$C(0,0)$	DC (block average)	Small	Almost always survives — dominates energy
$C(u,v)$ low $u+v$	Low frequency	Small/medium	Partially preserved
$C(u,v)$ high $u+v$	High frequency	Large	Often becomes zero — source of compression

5.13.3.4 📦 Input and Output Specification (VPL)

Input:

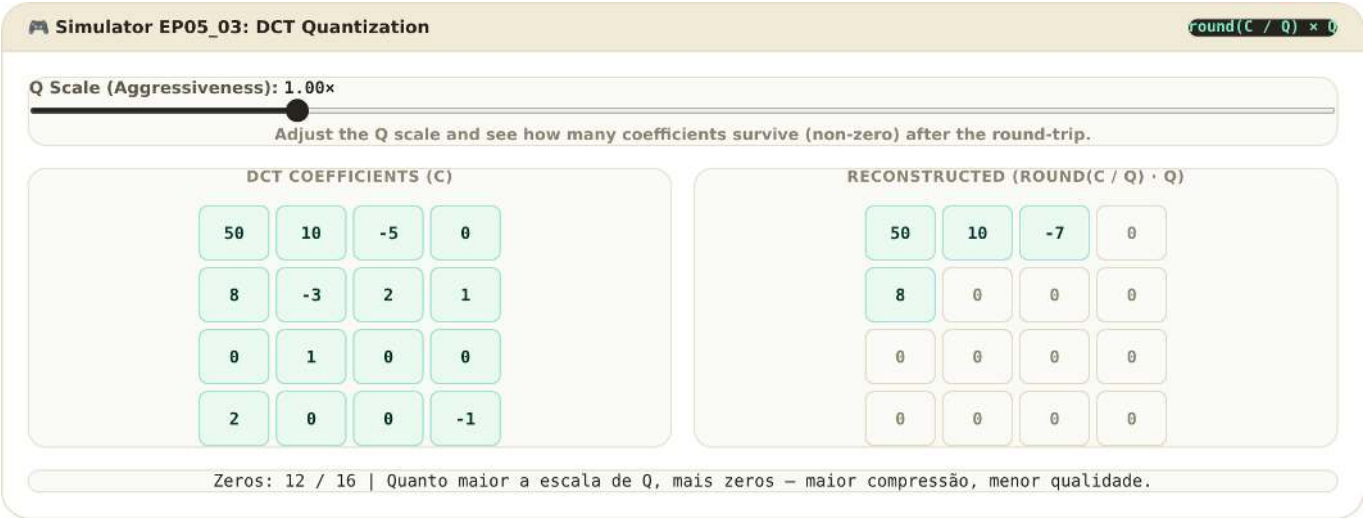
- Line 1: Integer N.
- The next N lines: matrix C (DCT coefficients, integers, may be negative).
- The next N lines: matrix Q (quantization table, positive integers).

Output:

- Reconstructed matrix C', $N \times N$, integers separated by spaces.

5.13.3.5 📌 Examples

Input	Output	Observation
4 50 10 -5 0 8 -3 2 1 0 1 0 0 2 0 0 -1 2 5 7 8 4 7 8 11 6 8 11 12 9 11 12 14	50 10 -7 0 8 0 0 0 0 0 0 0 0 0 0 0	$C(0,0) = 50/2 = 25 \to 25 \times 2 = 50$ (preserved). $C(0,2) = -5/7 \approx -0.71 \to -1 \to -1 \times 7 = -7$. Meanwhile, $C(1,1) = -3/7 \approx -0.43 \to 0$: zeroed by quantization — most of the block becomes zero, illustrating the energy compaction in the upper-left corner.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap05/sim-ep0503.html

Figure 5.34: EP05_03 Simulator: DCT Quantization (*round-trip*)

```
%%writefile EP05_03.cpp
// your solution
```

```
Overwriting EP05_03.cpp
```

```
TestSuite("EP05_03.cpp").run()
```

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

5.13.4 EP05_04 🔴 Implementing the 2D DFT from the Definition

A research laboratory in **computational astronomy** received, from an old mission, a small experimental sensor whose raw data cannot be processed by modern FFT libraries — the validation environment is isolated and only allows basic arithmetic operations. The team needs to **reimplement the 2D Discrete Fourier Transform from the mathematical definition itself**, cell by cell, to later compare bit by bit with `np.fft.fft2` in another environment.

This is the most conceptual exercise on the list: there are no shortcuts. You will implement the double summation from Equation 5.1 directly, demonstrating *why* the FFT exists — and the computational cost it avoids.

5.13.4.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers M (rows) and N (columns) of the image $f(x, y)$.
2. **Data:** Read the integer values of $f(x, y)$, row by row.
3. **2D DFT:** For each frequency pair (u, v) with $u = 0, \ldots, M-1$ and $v = 0, \ldots, N-1$, compute

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y)\, e^{-j2\pi\left(\frac{ux}{M} + \frac{vy}{N}\right)}$$

   using Euler's identity $e^{-j\theta} = \cos(\theta) - j\sin(\theta)$ to separate the real and imaginary parts — **do not use any ready-made FFT function**.
4. **Magnitude:** Compute $|F(u,v)| = \sqrt{\mathrm{Re}(F)^2 + \mathrm{Im}(F)^2}$ and round to the nearest integer.
5. **Output:** Display the matrix of rounded magnitudes, $M \times N$, in the same order (without `fftshift` — the DC remains at $(0, 0)$).

5.13.4.2 📌 Computational Constraints

- **Using FFT libraries is prohibited:** the implementation must compute the double summations explicitly (nested loops), even if slower.
- **No `fftshift`:** the output maintains the raw DFT convention, with the DC component at $F(0, 0)$ (upper-left corner).
- **Rounding:** the final magnitude must be rounded to the nearest integer; in the test cases there is no `.5` ambiguity.
- **Precision:** small floating-point errors (on the order of 10^{-6}) before rounding are expected and do not affect the final integer result.

5.13.4.3 🧠 Theoretical Foundation

Element	Meaning
$F(0,0)$	DC component — sum of all pixels, $F(0,0) = \sum f(x,y)$
Real part $\mathrm{Re}(F)$	Projection of the signal onto cosines
Imaginary part $\mathrm{Im}(F)$	Projection of the signal onto sines

Element	Meaning
Complexity of this implementation	$\mathcal{O}((MN)^2)$ — this is why the FFT, with $\mathcal{O}(MN\log(MN))$, is indispensable for real images

5.13.4.4 📦 Input and Output Specification (VPL)

Input:

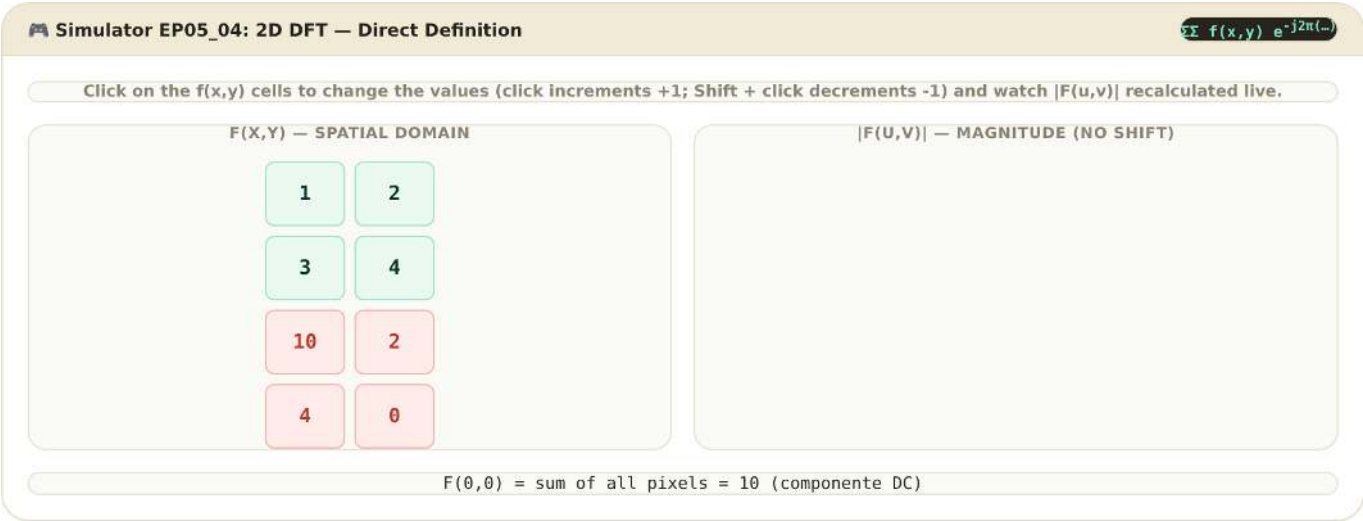
- Line 1: Integer M.
- Line 2: Integer N.
- Following lines: Integer elements of $f(x,y)$, M lines.

Output:

- Matrix of rounded magnitudes $|F(u,v)|$, $M \times N$, separated by spaces.

5.13.4.5 📌 Examples

Input	Output	Observation
2 2 1 2 3 4	10 2 4 0	$F(0,0) = 1+2+3+4 = 10$ (DC = total sum). $F(0,1) = (1-2)+(3-4) = -2 \to \|F\| = 2$. $F(1,0) = (1+2)-(3+4) = -4 \to \|F\| = 4$. $F(1,1) = (1-2)-(3-4) = 0$.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap05/sim-ep0504.html

Figure 5.35: EP05_04 Simulator: Manual 2D DFT

```
%%writefile EP05_04.cpp
// your solution
```

```
Overwriting EP05_04.cpp
```

```
TestSuite("EP05_04.cpp").run()
```

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

5.13.5 EP05_05 🏆 Complete JPEG Pipeline: DCT, Quantization, and Reconstruction

You have been hired to create, from scratch, a **didactic JPEG codec** in an embedded environment, without any available image library—only basic mathematical operations. The client wants to understand exactly where quality is lost and where it is recovered, block by block. This is the final challenge of the chapter: integrate **everything** that has been studied—the orthonormal DCT-II, perceptual quantization, and IDCT-based reconstruction—into a single end-to-end *pipeline*, processing an $N \times N$ block from start to finish, exactly as the JPEG standard does internally, 8×8 pixels at a time.

5.13.5.1 📋 Implementation Guidelines

1. **Block dimension:** Read the integer N.
2. **Original block:** Read the pixel matrix $f(x, y)$, N rows with N integers in $[0, 255]$.
3. **Quantization table:** Read the matrix Q, $N \times N$ positive integers.
4. **Centering:** Subtract 128 from each pixel: $g(x, y) = f(x, y) - 128$.
5. **Orthonormal 2D DCT-II:** Compute

$$C(u, v) = \alpha(u)\,\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} g(x, y)\, \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right]$$

   with $\alpha(0) = \sqrt{1/N}$ and $\alpha(k) = \sqrt{2/N}$ for $k > 0$.
6. **Quantization:** $\tilde{C}(u, v) = \text{round}(C(u, v)/Q(u, v))$.
7. **Dequantization:** $C'(u, v) = \tilde{C}(u, v) \times Q(u, v)$.
8. **2D IDCT-II (orthonormal inverse):** Compute $g'(x, y)$ from $C'(u, v)$ using the corresponding inverse transform (same basis, summation over u, v).
9. **Centering reversal and rounding:** $f'(x, y) = \text{round}(g'(x, y) + 128)$, restricted to the interval $[0, 255]$ (*clipping*).
10. **Output:** Display the reconstructed block f', $N \times N$, integers.

5.13.5.2 📌 Computational Constraints

- *__Complete pipeline mandatory:__ all six stages (center, DCT, quantize, dequantize, IDCT, revert) must be implemented—skipping quantization will not pass the tests, as the result would be identical to the original.
- *__Clipping:__ reconstructed values outside $[0, 255]$ must be truncated (0 if negative, 255 if greater than 255).
- **Rounding:** both in quantization and in final pixel reconstruction, use standard rounding; the test cases avoid `.5` ambiguity.
- **Orthonormal basis:** the normalization $\alpha(u)$ and $\alpha(v)$ must be applied exactly as specified—without it, the IDCT will not reconstruct correctly.

5.13.5.3 🧠 Theoretical Foundation

Stage	Analogous in the real JPEG standard	Where quality is lost
Centering	Same—DCT assumes a signal centered at zero	No loss
DCT-II	Steps 3–4 of the *pipeline* (Table 5.7)	No loss (exact and reversible transformation)
Quantization	Step 5—division by $Q(u, v)$	**Main source of loss**—high-frequency coefficients become zero
IDCT	Final reconstruction	Reconstructs exactly the *quantized* coefficients, not the original ones

5.13.5.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer N.
- Next N lines: original block $f(x, y)$, integers in $[0, 255]$.
- Next N lines: quantization table Q, positive integers.

Output:

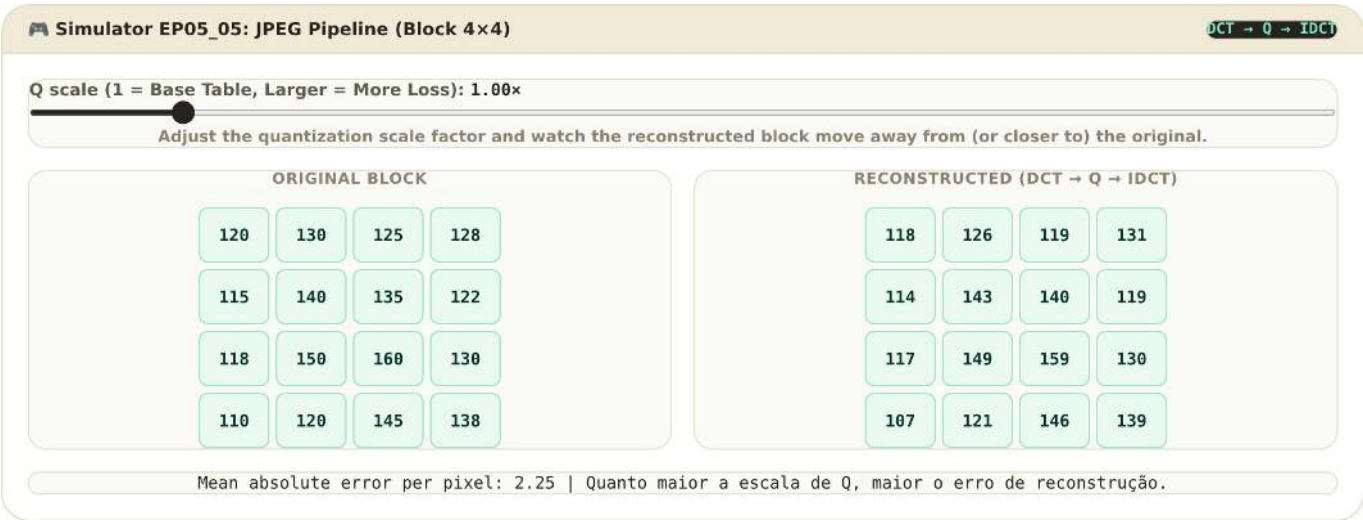
- Reconstructed block $f'(x, y)$, $N \times N$, integers in $[0, 255]$, separated by spaces.

5.13.5.5 📌 Examples

Input	Output	Observation
4 120 130 125 128 115 140 135 122 118 150 160 130 110 120 145 138 4 6 8 10 6 8 10 12 8 10 12 16 10 12 16 20	118 126 119 131 114 143 140 119 117 149 159 130 107 121 146 139	After DCT, aggressive quantization of high frequencies (large Q values in the lower-right corner) and IDCT-based reconstruction, the block remains **close** to the original but not identical—the difference is the cost of *lossy* compression.

5.13.5.6 💡 Debugging Tip

If the result does not match, check in this order: (1) the raw DCT coefficients (before quantization)—they should reconstruct the original **exactly** via IDCT if you skip steps 6–7; (2) the $\alpha(u)$ table—a common mistake is applying $\sqrt{2/N}$ also for $u = 0$; (3) the quantization rounding, which must occur **before** multiplying back by Q.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.en/cap05/sim-ep0505.html

Figure 5.36: EP05_05 Simulator: Complete JPEG pipeline in block

```
%%writefile EP05_05.cpp
// your solution
```

```
Overwriting EP05_05.cpp
```

```
TestSuite("EP05_05.cpp").run()
```

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

References

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.

ZAMPIROLLI, Francisco A. **MCTest: Como criar e corrigir exames parametrizados**. *[S.l.]*: Independente, 2023.

ZAMPIROLLI, Francisco de Assis *et al.* A practical digital image processing course with morph.py. *In*: Porto Alegre, RS, Brasil: Sociedade Brasileira de Computação (SBC), 2024. Disponível em: <https://sol.sbc.org.br/index.php/educomp/article/view/28199>

ZAMPIROLLI, Francisco de Assis *et al.* Teaching hands-on digital image processing with morph.py: Methods and comprehensive results. **Revista Brasileira de Informática na Educação**, v. 33, p. 990–1026, 2025.

ZAMPIROLLI, Francisco de Assis *et al.* Intelligent feedback for individualized introductory programming exercises. **Computer Applications in Engineering Education**, v. 34, n. 1, p. e70132, 2026.

Appendix A

About MCTest

MCTest is an open-source system for creating and automatically grading parameterized exams (Zampirolli, 2023). It supports multiple-choice, essay, and programming questions, the latter integrated with Moodle's VPL, described in Appendix B.

The version of MCTest used in this book is **5.4**, available at https://github.com/fzampirolli/mctest. The repository's `book` subfolder contains versions **5.3**, described in (Zampirolli, 2023), and **5.4**, used to generate the exams presented in this book and currently in production at UFABC (https://mctest.ufabc.edu.br).

A.1 Installing MCTest

The recommended installation uses VirtualBox with Ubuntu 22.04. In the Ubuntu terminal, as *root*, run:

```
wget https://raw.githubusercontent.com/fzampirolli/mctest/master/_setup-all.sh
```

Then replace `yourLogin` with the local username and run:

```
sed -i 's/\/home\/fz\//\/home\/yourLogin\//g' _setup-all.sh
source _setup-all.sh
pip install mysqlclient
```

After a few minutes, MCTest will be configured. To run it:

```
source /home/yourLogin/PycharmProjects/runDjango.sh
```

The system is then accessible at `http://127.0.0.1:8000`.

A.2 Creating an exam in MCTest

In MCTest, every exam is configured on an **Exam** screen, which brings together Classes, Topics (associated with a subject, such as PDI-VC), and Questions — each question belongs to a single topic. Besides the usual database attributes, the exam has its own parameters, such as the number of questions drawn and the number of variations generated.

For the two exams described in this appendix, a set of parameterized questions was defined per exam, from which a subset is randomly drawn for each variation, totaling **110 distinct variations** — one per student enrolled in the classes.

The **Create-Variations** button generates a new set of variations each time it is triggered. When questions are linked to VPL activities, the instructor receives by e-mail a `*linker.json` file containing all the test cases for the generated variations. The **Create-PDF** button generates, for each class, a PDF with the exams drawn per student, plus a `*students_variations.csv` file with each student's name, e-mail, and drawn variation.

> **!** Warning
>
> Each click of the **Create-Variations** button generates a new set of exam variations. After printing the PDF, it is essential **not to change the exam's attributes**, at the risk of invalidating the automatic grading of exams already printed or applied.

A.3 Creating a parametric question

A parametric question in MCTest combines three elements:

1. **Description** — the question statement, written in LaTeX, containing variables between `[[code:variable]]`, which are replaced by the value drawn for each student;
2. **Definition block** (`[[def: ... ]]`) — a Python code snippet, embedded in the question itself, responsible for drawing the parameters, computing the reference solution, and building the test cases;
3. **Test cases for Moodle** (`moodle_cases` block) — a JSON-serialized dictionary containing the expected inputs and outputs, consumed by the VPL activity.

A simplified example of a question definition (adapted from generating an `h × w` chessboard) is:

```
[[code:texto]]

\vspace{2mm}\noindent\textbf{Input Example:}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_inp]]
\end{verbatim}
\vspace{-2mm}\noindent\textbf{Output Example:}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_out]]
\end{verbatim}

\begin{comment}
[[code:moodle_cases]]
\end{comment}

[[def:
import json
import numpy as np
from topic.morph import mm # MUST include "topic." in MCTest

def chess(h, w):
    m = np.zeros((h, w), dtype='int')
    for i in range(h):
        for j in range(w):
            if (i + j) % 2:
                m[i][j] = 1
    return m

# PARAMETERS USED IN THE QUESTION DESCRIPTION
height = int(np.random.randint(5, 15))

# FULL STATEMENT, WITH THE height VALUE ALREADY INTERPOLATED
texto = (
    r"\vspace{2mm}\noindent\textbf{Description:}\vspace{-2mm} "
    r"Write a program that reads an integer \texttt{W}, representing the "
    r"width (number of columns) of a board, and prints that board in "
    r"the form of a chessboard, using the digits \texttt{0} and \texttt{1}. "
    r"The printed board will always have a height (number of rows) equal to "
    f"\\textbf{{{height}}} "
    r"and a width equal to the value \texttt{W} read from the input. "
```

```
    r"Considering position $(i, j)$ of the board (indexed from 0, with "
    r"$i$ representing the row and $j$ the column), the value printed at "
    r"that position must be \texttt{1} if $i + j$ is odd, and \texttt{0} otherwise. "
    r"Each row of the board must be printed on a separate line, with the "
    r"values of each column separated by a space."
)

inp_list, out_list, test_cases = [], [], 4
for i in range(test_cases):
    width = int(np.random.randint(500, 1500) / 100)
    inp = str(width) + '\n'
    out = mm.drawImg(chess(height, width)) + '\n'
    inp_list.append(inp)
    out_list.append(out)

cases = {}
cases['skills']      = ["matrices", "loops", "output formatting"]
cases['description'] = [{"text": latex_to_text(texto)}]
cases['input']       = inp_list
cases['output']      = out_list

moodle_cases = json.dumps(cases)

caso0_inp = cases['input'][0]
caso0_out = cases['output'][0]
]]
```

The `height` value is drawn once per variation (to appear in the statement), while `width` is drawn for each test case, increasing the robustness of automatic grading.

It is worth highlighting the dual role of the `texto` variable in this mechanism. It is at the same time:

- **the content of the printed statement**, inserted into the question's LaTeX description through the `[[code:texto]]` tag, appearing in the PDF generated by the **Create-PDF** button; and
- **an entry in the dictionary exported to Moodle**, through the `cases['description']` key, which becomes part of the `moodle_cases` JSON. It is precisely this field that the VPL activity shows the student when they open the question in Moodle to review it or submit a solution.

There is, however, an important difference between these two uses: the PDF is composed in LaTeX and therefore normally interprets commands such as `\textbf{}`, `\texttt{}`, or `$...$`. Moodle's VPL, on the other hand, does **not** process LaTeX — it expects plain text. That is why, when building `cases['description']`, `texto` is not inserted directly, but passed through MCTest's own utility function `latex_to_text()`, which removes (or converts) the LaTeX commands and symbols from the statement, producing an equivalent plain-text version. Thus, `[[code:texto]]` in the PDF still receives the original `texto`, with all its LaTeX formatting, while `cases['description']` receives `latex_to_text(texto)`, ensuring that the statement shown in Moodle is readable even without LaTeX support.

Since `texto` is built inside the `[[def: ...]]` block itself — in the same scope where `height`, `width`, and the other parameters are drawn —, it is recalculated for every variation. This guarantees that the statement seen by the student in Moodle is **always identical** to the one printed on the paper exam, even when the text changes from student to student along with the drawn values. Section A.4 revisits this point with a real case, in which the statement is significantly longer and describes, besides the numeric parameters, the visual scenario itself generated for each variation.

Figure A.1 shows the chessboard question actually generated by MCTest for one of the variations, with the statement already containing the drawn value of `height` and the input/output example corresponding to the first test case:

Clicking **Create-PDF** on the question screen generates a new variation with each click, allowing the statement to be checked before publishing.

1. **Descrição:** Escreva um programa que leia um número inteiro `W`, representando a largura (número de colunas) de um tabuleiro, e imprima esse tabuleiro no formato de um tabuleiro de xadrez, usando os dígitos `0` e `1`. O tabuleiro impresso terá sempre altura (número de linhas) igual a **6** e largura igual ao valor `W` lido da entrada. Considerando a posição (i, j) do tabuleiro (indexada a partir de 0, com i representando a linha e j a coluna), o valor impresso nessa posição deve ser `1` se $i + j$ for ímpar, e `0` caso contrário. Cada linha do tabuleiro deve ser impressa em uma linha separada, com os valores de cada coluna separados por um espaço.

Exemplo de Entrada:

```
9
```

Exemplo de Saída:

```
0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1
```

Figure A.1: Chessboard question generated by MCTest, illustrating the parameterized statement with the `height` value drawn for the variation and the input/output example of the first test case.

For the two exams of the course, this mechanism was used more broadly: each parametric question defines not only numeric values but also, in some cases, small structural variations in the statement (for example, which cardinal direction — North, South, East, West — should be evaluated), making it harder to find ready-made solutions online or via generative AI tools.

A.4 Real example: a question from Mock Exam 4

To illustrate the mechanism described in the previous section with a concrete case, the following is a question actually applied in **Mock Exam 4** (topic *im4-Morphological Operators*, difficulty 1, Bloom's taxonomy "remember"), a parameterized programming question with Moodle+VPL integration.

The question asks the student to write a program capable of:

- reading a binary image, corrupted by salt-and-pepper noise, containing at least two isolated geometric objects;
- applying morphological filtering (opening followed by closing) to remove the noise;
- extracting, from the filtered image, the geometric measurements of each object (area, perimeter, center, bounding box, circularity, solidity, and number of vertices), using the helper function `mm.measure`;
- sorting the objects by position (bounding-box X coordinate, with Y as tiebreaker) and reassigning the identifiers sequentially;
- printing the cleaned matrix and the measurement table in the format expected by the automatic grader.

In the corresponding `[[def: ... ]]` block, a scene generator (`gerarCena`) randomly draws the height and width of the image, the type, size, and position of each object (square, rectangle, triangle, or block), ensuring they do not overlap, and then adds salt-and-pepper noise with a 3% probability per pixel. Ten distinct test cases are generated this way for each exam variation, and the input/output pair of the first case is reused in the statement itself as an example for the student. The morphology library (`mm`) is imported directly from `morph.py`, also available as MCTest's own utility module.

In summary, the statement asks the student for a program that:

1. reads, on the first line, two integers H and W (image height and width);
2. reads the following H lines of the binary matrix (0s and 1s separated by spaces), using `mm.readImg(H, W)`;
3. applies morphological filtering to remove the salt-and-pepper noise;
4. prints the already filtered matrix, as 0s and 1s separated by spaces;
5. extracts the geometric measurements of the objects with `mm.measure(imgLimpa)`;
6. sorts the objects and prints the measurement table in the expected format.

The statement also brings two important observations for automatic grading: (i) the area computed by

OpenCV (`cv2.contourArea`) corresponds to the continuous polygon delimited by the centers of the border pixels, and is therefore smaller than the simple count of pixels equal to 1 (`np.sum`); and (ii) the list of objects must be sorted in ascending order by the bounding-box X coordinate (`bbox[0]`), using the Y coordinate (`bbox[1]`) as tiebreaker, with the IDs reassigned sequentially from 1 to N after sorting.

As in the example in Section A.3, the statement text here is also not fixed: it is built inside the `[[def: ... ]]` block itself, in the Python variable `texto`, and plays the same dual role already described — it feeds the PDF via `[[code:texto]]` and is exported, already converted by `latex_to_text()`, into `cases['description']` inside `moodle_cases`, which is the version shown to the student in Moodle's VPL. The difference is that, in this real case, it is the **visual scenario** (noisy image, number and position of objects) that changes from student to student with each variation, while the wording of `texto` stays fixed across variations, since only the numeric data (image and test cases) are drawn. Ideally, the wording would also vary with each generation, as in the previous example, where the `height` value was interpolated directly into the description text. The actual structure used (with scenario-drawing snippets omitted for brevity) is:

```
[[code:texto]]

\vspace{2mm}\noindent\textbf{Input Example:}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_inp]]
\end{verbatim}
\vspace{-2mm}\noindent\textbf{Output Example:}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_out]]
\end{verbatim}

\begin{comment}
[[code:moodle_cases]]
\end{comment}

[[def:
import json
import numpy as n
import cv2
from topic.morph import mm # MUST include "topic." in MCTest

# ... scene, noise, and 10 test-case generation (inplist/outlist) ...

# FULL STATEMENT, WITH EXPLANATION OF AREA AND SORTING
texto = (
    "A binary image corrupted by salt-and-pepper noise contains at least two isolated
    geometric objects.\n\n"
    "Write a program that:\n"
    "1. Reads two integers ..."
)

cases = {}
cases['skills']      = ["mathematical morphology", "noise removal", "mm.measure",
"tabulation"]
cases['description'] = [{"text": latex_to_text(texto)}]
cases['input']       = np.array(inplist).tolist()
cases['output']      = np.array(outlist).tolist()

moodle_cases = json.dumps(cases)
caso0_inp = cases['input'][0]
caso0_out = cases['output'][0]
]]
```

Below is the photograph of the question actually generated and printed for one of the 110 variations of Mock Exam 4, showing the statement, the noisy input image, and the expected output example (filtered image and

measurement table):

1. Uma imagem binária corrompida por ruído sal e pimenta contém no mínimo dois objetos geométricos isolados.

Escreva um programa que: 1. Leia dois inteiros na primeira linha representando a altura (H) e largura (W) da imagem. 2. Leia as H linhas seguintes da matriz binária (valores 0 e 1 separados por espaço) usando mm.readImg(H, W). 3. Aplique a filtragem morfológica para eliminar o ruído sal e pimenta. 4. Imprima a matriz limpa diretamente em valores 0 e 1 (0s e 1s separados por espaço). 5. Extraia as medições geométricas dos objetos usando mm.measure(imgLimpa). 6. Ordene a lista de objetos e imprima a tabela formatada das medidas.

Observação sobre a área: A área calculada pelo OpenCV (cv2.contourArea) mede o polígono contínuo delimitado pelos centros dos pixels de borda, resultando em um valor menor do que a simples contagem discreta de pixels 1 (np.sum).

Regra de Ordenação: A lista de objetos deve ser ordenada em ordem crescente pela posição X do retângulo delimitador (bbox[0]), utilizando a posição Y (bbox[1]) como critério de desempate caso dois objetos comecem na mesma coluna (medidas.sort(key=lambda m: (m['bbox'][0], m['bbox'][1]))). Em seguida, reatribua os IDs sequencialmente de 1 a N.

Exemplo de Entrada:

```
12 43
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 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 0 1 1 1 1 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 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 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 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 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 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 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 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 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 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 1 1 1 1 1 1 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 1 1 1 1 1 1 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 1 1 1 1 1 1 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 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 id area perimeter     cx     cy   x   y   w   h circularity solidity vertices
  1   20      18.0    6.5    2.0   4   0   6   5        0.78    1.000        4
  2   20      18.0   26.0    2.5  24   0   5   6        0.78    1.000        4
```

Figure A.2: Real question generated by MCTest for Mock Exam 4 — “Morphological Operators” topic.

A.5 Final remarks

This appendix summarized the steps to install MCTest, create exams, and build parametric questions with Moodle VPL integration. The combined use of variables drawn in the description (`[[code:...]]`) and Python code embedded in the question itself (`[[def: ... ]]`) makes it possible to generate hundreds of exam variations from a single template, each graded automatically and consistently — with the `texto` variable, in particular, guaranteeing that the same statement is shown both in the printed PDF and when the question is reviewed by the student in Moodle. The real example of Mock Exam 4 (Section A.4) illustrates how this mechanism is used in practice to generate programming questions with visual scenarios that differ per student. Appendix B details how these variations are published and graded in Moodle, and Appendix C describes how to restrict access to exams using SEB.

Appendix B

About Moodle (VPL)

This appendix describes the configuration of a **VPL (*Virtual Programming Lab*)** activity in Moodle, the *plugin* used for submitting and automatically grading the Programming Exercises (EPs), the biweekly mock exams, and the exams described in this book. The Moodle version used was **4.5**.

B.1 Configuring a VPL activity

Creating an exam in MCTest with VPL integration (see Appendix A) generates two files that must be renamed to `linker.json` and `students_variations.csv`. These files, together with `morph.py`, must be added to the VPL activity's **Execution Files**, along with all the files contained in `VPL_modification/V14-AI-feedback.zip`, available in the MCTest repository at https://github.com/fzampirolli/mctest. All of them must be marked with the **"*Files to keep when running*"** option.

In short, the VPL activity then knows, for each student, which variation of the question was drawn (via `students_variations.csv`) and which set of test cases corresponds to it (via `linker.json`). This allows grading to be individualized, even when dozens of students solve the same activity with different variations.

B.2 Publishing EPs as VPL activities

The Programming Exercises (EPs) presented at the end of each chapter of this book can also be published as VPL activities, following the same mechanism. The typical classroom usage flow was:

1. Opening the chapter's two Colab *notebooks* (theoretical and practical), with emphasis on the interactive simulators;
2. Running the code blocks, changing parameters to reinforce understanding of the concepts;
3. In lab classes, submitting the EP answers directly in the corresponding VPL activity, with immediate feedback on passing or failing the test cases.

If the VPL activity is not generated by MCTest, the `linker.json` and `students_variations.csv` files are not needed. However, to use AI *feedback* (see Appendix D), the files contained in `VPL_modification/V14-EP0_VPL-TEMPLATE.zip`, available in the same repository (https://github.com/fzampirolli/mctest), must be added, following the procedure described in Section B.1.

> **Reusing test cases**
>
> The `.cases` files used in local validation (the `TestSuite` class, described in the body of this book) are compatible with the format expected by VPL, allowing the same set of tests to be reused both in EP development and in automated grading in Moodle.

In the biweekly mock exams — applied with SEB (see Appendix C) and worth a 5% bonus on the final grade —, VPL activities were used with EPs similar to those in the regular assignment lists, but graded under internet-access restrictions, reinforcing the assessment nature of the activity.

B.3 Final remarks

This appendix described the configuration of VPL activities in Moodle for automatically grading EPs, mock exams, and parameterized exams. The combination of MCTest (generating variations and test cases), VPL (submission and grading), and SEB (access restriction) forms the tripod on which the automated, individualized assessment used throughout the course rests.

Appendix C

Using SEB in Biweekly Mock Exams and Assessments

Besides the two exams with parameterized questions (see Appendix A), the ***Safe Exam Browser*** **(SEB)** was also used to restrict access during the **biweekly mock exams** — VPL (*Virtual Programming Lab*) activities with Programming Exercises (EPs) similar to those in the regular assignment lists, worth a bonus of up to 5% on the final grade —, as well as the other practical assessments.

SEB ensures a secure assessment environment by blocking access to unauthorized external resources on the lab machines. For the implementation adopted in this work, **Windows 10** and SEB version **3.7.1.704**, available at https://safeexambrowser.org/, were used, and must be pre-installed on every lab workstation.

The procedure for configuring the SEB environment file and linking it to the VPL activity in Moodle is described below.

C.1 Configuring the application in *SEB Tool*

To generate the `.seb` configuration file that will be distributed to students, open the *SEB Tool* utility and follow the steps below.

1. ***General***
   - In the ***Start URL*** field, enter the direct URL of the VPL activity in Moodle.
   - *Note:* if it is necessary to navigate to more than one activity within the same course, enter the course's main URL and, on the ***Browser*** tab, check the ***Allow navigation back/forward in exam*** option.
2. ***Network***
   - Include the URLs allowed for navigation. To restrict access to the desired activity, use a filter expression in the pattern `/mod/vpl/*?id=4624*`, where the final number corresponds to the VPL activity's identifier in Moodle. In Table C.1, the last row contains the URL of the course's main page (identifier `475`), but it can also be replaced by the URL of a section containing several activities. This setting is optional and useful when there is more than one VPL activity. In that case, the first row of the table should be repeated for each assessment activity that should remain accessible during the *exam*.
   - **General settings on the *Filter* tab**:
     ☒ *Activate URL filtering*
     ☐ *Filter also embedded content*

Table C.1: URL filter expressions used in the SEB configuration.

Active	*Regex*	*Expression*	*Action*
[x]	[ ]	`https://moodle.ufabc.edu.br/mod/vpl/*?id=4624*`	*Allow*
[x]	[ ]	`https://moodle.ufabc.edu.br/login/index.php*`	*Allow*
[x]	[ ]	`https://moodle.ufabc.edu.br/course/switchrole.php*`	*Allow*
[x]	[ ]	`https://mctest.ufabc.edu.br/compare`	*Allow*

Active	*Regex*	*Expression*	*Action*
[x]	[ ]	`https://moodle.ufabc.edu.br/enrol/index.php?id=475`	*Allow*

3. ***Security***
   - If the lab uses a projector or a secondary monitor, set the ***Maximum allowed number of connected displays*** field to a value greater than 1, to prevent SEB from blocking the display.
4. ***Config File***
   - Click ***Save Settings As...*** and save the file with the `.seb` extension.
5. ***Exam***
   - Check the ***Use Browser Exam Key and Configuration Key*** option.
   - Copy the code generated in the ***Browser Exam Key*** field. *Important:* copy this key only **after** saving the file in the previous step.

C.2 Linking the SEB key to the VPL activity in Moodle

After generating the file and obtaining the *Browser Exam Key*, access Moodle to apply the restrictions.

1. On the VPL activity page, click the gear icon (**Settings**).
2. Go to ***Configuration*** → ***Submission restrictions*** and click ***Show more***.
3. Change the ***SEB browser required*** option to ***Yes***.
4. Paste the copied key into the ***SEB exam key(s)*** field.

C.3 Restriction by IP address (optional)

To ensure that submissions come exclusively from the lab computers, SEB can be combined with Moodle's network filter, by filling in the ***Allowed submission from net*** field.

- **Consecutive ranges:** use a hyphen to define the range. Example: `111.11.11.1-62` (allows IPs from `111.11.11.1` to `111.11.11.62`).
- **Non-consecutive IPs:** list the individual addresses separated by commas.

C.4 Final remarks

The integration of SEB with VPL in Moodle creates a simple flow for the student: just double-click the `.seb` file provided by the instructor for the secure browser to open directly on the configured activity.

The combination of the *Browser Exam Key* with IP-range restriction on the lab machines provides a robust solution for *online* programming assessments, hindering improper access and contributing to the integrity of the assessment process.

Appendix D

Generating Course Material with *Gemini Notebook*

This appendix describes the use of ***Gemini Notebook*** **(*NotebookLM*)** as a tool supporting the preparation of course material, complementing the use of AI for reviewing text and code, mentioned in this book's preface.

D.1 Conceptual videos and EP-solving videos

For each chapter, a project was created in *Gemini Notebook* using the full PDF of the book and the `morph.py` file as sources. From these sources, *Gemini Notebook* automatically generated a short video (7 minutes on average), used at the start of classes. Chapters alternated between two types of video:

- **Conceptual video**, presenting the chapter's theoretical foundations, generally shown in the first class dedicated to the topic;
- **EP-solving video**, presenting a solution strategy for the Programming Exercises, shown in the following class, when most EPs were solved together with the class.

> **i** Role of the video in class
>
> The video generated by *Gemini Notebook* does not replace the instructor's explanation. It works as a brief introduction and motivation for the topic, giving the student context before the *slides* are presented and the Colab *notebooks* are opened.

D.2 Supporting *slides*

Complementing the videos, *Gemini Notebook* also generated, from the same sources (the book's PDF and the `morph.py` file), a set of approximately 12 *slides* per chapter, covering both the theoretical concepts and the practical part. Generation used a chapter-specific *prompt*, restricting the scope of content to be synthesized and avoiding both anticipating topics from later chapters and reproducing long passages of the book without didactic adaptation.

After the *slides* were presented, class continued with the opening of the chapter's two Colab *notebooks* (theoretical and practical), emphasizing the interactive simulators and running the code blocks with parameter changes.

D.3 Main material and final remarks

Generating videos and *slides* with *Gemini Notebook*, based on the book itself and the `morph.py` library, made it possible to standardize the opening of classes and reduce the time spent preparing course material, without dispensing with the instructor's curation in crafting the *prompts* and conducting classroom activities.

The videos and *slides* produced by *Gemini Notebook* are **motivational** in nature and serve as a starting point for discussing the concepts presented in each chapter. Like any AI-generated content, they may contain

occasional inaccuracies or errors. In some cases, these inaccuracies are intentionally exploited during class to check whether students are critically following the presentation and can identify conceptual inconsistencies.

The course's official content, however, is the material produced by the method described in this book's preface, made available in three complementary formats: **HTML**, for browsing with interactive simulators; **PDF**, for reading and printing; and **Jupyter** *notebooks* (`.ipynb`), for running and experimenting with the code. All material generated by *Gemini Notebook* should be understood as a complement to this main content.

As with the other uses of generative AI described in this book, responsibility for the technical quality, conceptual correctness, and pedagogical adequacy of the material remains with the instructor.

Appendix E

Using AI in Student Assessment: the vpl-ai-feedback Project

This appendix describes **vpl-ai-feedback**, an open-source system developed to support the grading of code submissions sent to VPL (Moodle) through *feedback* generated by Artificial Intelligence (AI). Unlike a traditional automatic grader — which only compares outputs against test cases —, **vpl-ai-feedback** sends the student's code, together with the question rubric, to a language model (LLM), which returns a qualitative analysis per criterion, in the style of ***Socratic* feedback**: the student receives comments pointing out what was done correctly, what can be improved and why, in both **formative** activities (practice exercises) and **assessment** activities (mock exams) in VPL.

This Socratic *feedback* mechanism for VPL activities was proposed by Zampirolli et al. (2026). The study describes the integration of AI into the assessment process for parameterized programming exercises in VPL/Moodle, using a set of seven LLMs adopted to analyze student code and generate automated *feedback* on every grading request, with preliminary results indicating a positive student perception regarding idea generation, clarity of explanations, and learning autonomy.

The rest of this appendix details the practical implementation of these ideas in the **vpl-ai-feedback** project — whose code is available at https://github.com/fzampirolli/vpl-ai-feedback — and specifically describes its use in the three PDI-VC classes between May and August 2026.

> ⚠ Warning: AI does not replace instructor judgment
>
> Using AI to **grade** students **is not recommended as the sole source of a grade**, since language models can **hallucinate** — that is, produce plausible but incorrect assessments, criteria, or justifications, even for correct or incorrect code. An LLM may give a high grade to code with a subtle bug, or a low grade to correct code written in an unusual way, simply because the generated explanation "sounds" coherent. **AI should be used only as a complement to assessment**, never as a substitute for the instructor. Appendix A and Appendix B describe the objective grading mechanisms (test cases) that remain the basis of the official grade; this appendix deals exclusively with the use of AI as an additional layer of qualitative *feedback*.

E.1 Context of use

vpl-ai-feedback was used in three sections of the **PDI-VC** course, taught between May and August 2026, totaling approximately one hundred students. AI *feedback* was used in three complementary ways:

1. **Continuous *feedback* in formative activities** — practice exercises submitted to VPL throughout the course, in which the student could resubmit code as many times as desired. On each submission, they received qualitative AI-generated *feedback*, in addition to the objective grading performed by VPL, described in Zampirolli et al. (2026).
2. **Complementary *feedback* in mock exams (bonus assessment activities)** — four mock exams held throughout the term, planned in the syllabus as bonus activities, each worth up to 5% of the final grade. Applied approximately 30 minutes before the end of the biweekly lab practical classes, the mock

exams used **vpl-ai-feedback** to generate an individual assessment rubric, sent by e-mail together with a comparative summary between the grade assigned by Moodle/VPL and the assessment produced by the AI.

3. **Complementary *feedback* on Exams 1 and 2 (official assessments)** — the term's two formal exams, each worth a larger share of the course's final grade, also began using **vpl-ai-feedback** after they were applied. Unlike the mock exams (bonus), here the official grade is already determined by VPL's objective grading and/or the instructor's manual review before results are released; the AI-generated rubric is sent to the student only afterward, as supporting material for reviewing their own performance — reinforcing, also in the higher-weight assessments, the separation between *official grade* and *complementary feedback* detailed in Section D.7.

An opinion survey, answered by 102 students before the system was adopted, indicated high acceptance of the proposal. Only 2 students gave a rating of 1 (rejection) and 7 gave a rating of 2 regarding interest in receiving automatic AI code *feedback*. Another 19 declared themselves indifferent (rating 3), while the majority — 38 and 37 students — gave ratings of 4 and 5 (high approval), respectively, totaling the 102 respondents. This result motivated the adoption of the system as a complement to the formative process, not as a substitute for human grading.

! The official grade was never the AI's grade

It is essential to stress that, as defined in the syllabus, **the grade used both for calculating each mock exam's bonus and for the official grade of Exams 1 and 2 was always the grade assigned by VPL** (based on the test cases) and/or by the instructor's manual review — and **not** the grade suggested by the AI. The rubric generated by **vpl-ai-feedback** was sent to the student only as supporting learning material — never as a criterion for assigning a grade, whether in bonus activities or official assessments. This separation between *official grade* (VPL/instructor) and *complementary feedback* (AI) is the central principle of this appendix and was revisited in Section D.8.

E.2 Project architecture

vpl-ai-feedback is an asynchronous Python *script* organized as follows:

```
.
  config.yaml              ← configuration (credentials, provider, weights) - NEVER versioned
  config.yaml.example      ← public configuration template
  main.py                  ← main script (async), orchestrates the whole process
  run.sh                   ← execution wrapper (bash run.sh config.yaml)
  gerar_relatorio.py       ← converts the consolidated *_ALL.txt into CSV
  enviar_email.py          ← sends the rubrics by e-mail to each student
  providers/                ← LLM API clients
      __init__.py          ← client factory (Factory)
      base.py               ← retry, backoff, and fallback logic between models
      groq.py
      deepseek.py
      gemini.py
  core/                      ← business logic
      grader.py             ← identifies files by question, builds the prompt, calls the LLM
      utils.py
  <class_folder>/          ← submissions downloaded from Moodle VPL
      Student name - login/
          <timestamp>/                ← student's latest submission
              Q1.py                   ← pattern for exams with multiple questions
              Q2.py
              rubrica.txt             ← generated by the LLM (only if the assessment is
   valid)
          <timestamp>.ceg/
```

```
            execution.txt            ← objective grade assigned by VPL
    ...
```

The system accepts three LLM providers — **Groq**, **DeepSeek**, and **Gemini** — configurable in `config.yaml` via the `llm.provider` key. Each provider can have **multiple registered models**; on each call, the list is **shuffled**, distributing the load among students processed in parallel and reducing the chance of rate-limit errors (HTTP 429). When a model fails, the system retries up to three times before moving to the next one in the list, respecting the wait time suggested by the provider. Authentication errors (401) or insufficient-balance errors (402) immediately abort the attempt with that provider.

An important aspect of the project concerns data privacy. **The student's name, e-mail, login, student ID, and national ID are never sent to the LLM API**. Only the following are transmitted for processing: the name of the file containing the source code; the code itself (with comments removed); the rubric *prompt* — which contains no personal data —; and, when available in the `execution.txt` file generated by VPL, the official statement of the question drawn for that student and the raw result of the test cases actually executed (input, output produced by the code, and expected output). These last two items, although extracted from a file specific to each student, also contain no personal identification — only the question text and the input/output values of the automated tests —, and are used as objective evidence to reduce the risk of the AI incorrectly identifying the question type or assessing the code's logic based solely on a static reading (Section D.3).

Even so, it is important to recognize that the three currently supported providers — Groq, DeepSeek, and Gemini — are commercial services operated by foreign companies, whose models run on infrastructure outside the country and are subject to usage, availability, and cost policies beyond the institution's control. This dependence on external providers carries risks that go beyond one-off code grading: price changes, model discontinuation, temporary service unavailability, unilateral changes to terms of use, and, in more sensitive cases, geopolitical access restrictions could compromise the continuity of a grading system that becomes structurally dependent on these APIs. For this reason, it is strategic for Higher Education Institutions (HEIs) such as UFABC to invest in developing and hosting their **own AI providers** — for example, open-weight models run on local infrastructure or a sovereign cloud —, reducing dependence on external services, especially from other countries, and expanding institutional control over cost, availability, student data privacy, and long-term pedagogical continuity. The design of **vpl-ai-feedback** already favors this path: the `providers/` architecture was built as an extensible factory, in which a new provider — including a model hosted locally by the institution itself, with a compatible interface — can be added with low integration effort.

E.3 The universal *prompt* and the two-stage rubric

Each exam type is described in a *prompt* file (for example, `Simulado4.txt`), referenced in `grading.prompt_file` in `config.yaml`. This file instructs the LLM to perform the assessment in **two stages**:

1. **Question identification** — the LLM determines the specific operation type drawn for that student (useful when there are parameterized questions drawn and shuffled per student, as described in Appendix A);
2. **Applying the corresponding rubric** — the LLM assesses the code against scored criteria, each with a maximum point range, ultimately returning a grade in the format `FINAL GRADE: X/WEIGHT`.

For parameterized exams, the *prompt* file usually contains **several parallel rubrics**, one for each possible question type, delimited by markers such as `[START_RUBRICA_TIPO_A]` / `[END_RUBRICA_TIPO_A]`. The LLM first identifies which type was drawn for that student and applies **only** the corresponding rubric, ignoring the others — which prevents criteria from a different type from contaminating the grade.

When the exam has more than one question per VPL activity, the code files must follow the pattern `Q1.*`, `Q2.*`, etc. — one file per question, with the extension free according to the language chosen by the student —, and each question has an individually configurable weight in `grading.weights` in `config.yaml`. When the exam has a single question and was not generated by MCTest, the system accepts any file name with a supported extension, as long as it is the only code file present in the submission folder.

In the case of Mock Exam 4 (topic *Morphological Operators*), for example, one of the possible types defined three criteria, with a total weight of 100 points for that question:

Criterion	Description	Maximum score
1 — Data input	Reading `H`, `W`, and the binary matrix (`mm.readImg`)	25 pts
2 — Morphological processing	Opening + Closing (`mm.sebox()`), `mm.measure`, sorting by `bbox[0]`/`bbox[1]`, and reassigning IDs	50 pts
3 — Output and formatting	Displaying the cleaned image (`mm.drawImg`) and the formatted measurement table	25 pts

The *prompt* also requires a **mandatory response format** (line width, section order, the `FINAL GRADE:` marker), which makes automatic grade extraction and the consolidated report generation more reliable — although, as discussed in Section D.8, this does not eliminate the risk of semantic error in assessing the content.

E.4 Two additional layers of evidence

To reduce the risk of the LLM hallucinating the question type or the correctness of the code — the central risk discussed in Section D.7 —, **vpl-ai-feedback** started providing the AI with two sources of objective evidence, automatically extracted from VPL's own `execution.txt`, in addition to the source code:

- **Official question statement.** Since questions are usually drawn and shuffled per student in MCTest (Appendix A), each student's `execution.txt` already contains, in the "Question summary" field, the exact text of the question they were assigned. The system automatically extracts this excerpt and sends it to the LLM as **primary** evidence for identifying the question's type/variation — more reliable than inferring solely from a static reading of the code, especially in incomplete submissions or those ambiguous between two close types (for example, "flat" erosion *versus* "weighted" erosion). When the submitted code diverges from what the statement asks, the system does not ignore the statement nor silently "correct" the reading: it keeps the identification based on the statement, lowers the assessment's confidence to "Low," and makes the divergence explicit in the *feedback* sent to the student.
- **Actual VPL execution result.** When available, the system also sends the LLM the test cases actually run on that code — input, output produced, and expected output — as objective evidence of correctness, to be used to confirm or refute the reading of the logic before scoring the processing and output criteria, instead of the AI relying solely on mentally simulating the code (which is especially error-prone in loops with `min`/`max`, index offsets, or Otsu threshold computation, where subtle errors go unnoticed in a static reading).

In addition, the LLM is instructed to explicitly state its **confidence level** (High, Medium, or Low) in identifying each question type, with justification when confidence is not High. This value is automatically extracted and feeds a `Revisar_Manualmente` (Review Manually) column in the consolidated CSV report (Section D.4), allowing the instructor to prioritize exactly the cases where the AI itself recognizes greater uncertainty — instead of treating all of a class's assessments as equally reliable.

E.5 Execution flow

The typical run, done via `bash run.sh config.yaml`, follows these steps:

1. Loads `config.yaml` and locates the submissions folder (`paths.student_base_dir`);
2. For each student, identifies the **latest submission** (the folder with the most recent *timestamp*);

3. Extracts the objective Moodle grade from `*.ceg/execution.txt`;
4. For each question of the exam, locates the code file (`Q1.*`, `Q2.*`, or the single file, for an exam not generated by MCTest), builds the *prompt* (code + rubric), and sends it to a model drawn from the configured list;
5. Processes all students **in parallel**, with concurrency control;
6. Generates, per student, the `rubrica.txt` file — **only if the AI returns at least one valid assessment** for the questions that have code; otherwise, the file is not saved, forcing a retry on the next run;
7. Consolidates all `rubrica.txt` files into a single `*_ALL.txt` and generates a `*_relatorio.csv` report, comparing the Moodle grade, the AI grade, and the difference between them, per question and in total.

Situation	Is `rubrica.txt` saved?
All questions with code successfully assessed	✅ Yes
AI failed on at least one question with code	❌ No — reprocessed on the next run
Student submitted no code file	❌ No
Question type identified as UNKNOWN	❌ No

E.6 Illustrative example: rubric for a three-question exam

i About the origin of this example

The following excerpt **does not reproduce any specific student's code**. It is an example reconstructed by the author, reproducing an error *pattern* repeatedly observed across PDI-VC classes — confusion between a 2D morphological function (`mm.ero`/`mm.ero0`) and a 1D operation on signals — without citing or identifying any actual submission. This choice avoids any risk of infringing a student's copyright over their own source code, even anonymized, since authorship of the work remains with the original author even when name and *login* are removed.

The following excerpt illustrates the output generated by **vpl-ai-feedback** for a student on a three-question exam (Q1, Q2, Q3), using the `deepseek-chat` model. The comparative summary appears at the top of the file, followed by the official statement of each question, the submitted code, and the assessment per criterion.

```
SUMMARY - AI (deepseek-chat)  x  MOODLE

Total weight : 100 pts

Q1 (AI) :  3.3 / 33 pts
Q2 (AI) :  6.7 / 33 pts
Q3 (AI) : 33.3 / 34 pts

Moodle : (Q1=33 + Q2=0 + Q3=34) = 67 pts
AI     : 43.3 / 100 pts

Difference (AI - Moodle): -23.7 pts

Q1: identified type = B  | confidence = Low (code does not
  match the question statement)  REVIEW
Q2: identified type = A  | confidence = High (official statement
  confirms the type and variation)
Q3: identified type = C1 | confidence = High (official statement
  confirms 2D erosion with integrated OpenCV)
```

Q1 — code does not match the statement (low confidence): the question's official statement,

extracted from `execution.txt`, asked for a **1D** morphological erosion on a signal (reading N, M, the signal, and structuring element B, followed by the minimum of the active neighbors). The submitted code, however, only reads two integers and calls `mm.ero0` — a **2D** erosion function for images, unrelated to the requested operation.

```
Identified type: TYPE B (1D Erosion - flat)
Confidence: Low - code does not match the question statement (uses
mm.ero0, a 2D erosion function, instead of implementing flat 1D
erosion on a signal)

Criterion 1 - [3.33/10.00 pts]:
- The code reads two integers on a single line (which would correspond
  to N and M), but does not read the signal or the structuring element
  B as 1D vectors. Instead, it treats both as images via mm.readImg,
  which does not match the expected reading of a 1D signal and of B on
  separate lines.

Criterion 2 - [0.00/16.67 pts]:
- The operation applied is mm.ero0(img, element), a 2D erosion for
  images, not the 1D signal erosion requested in the statement. The
  logic of the minimum of active neighbors with border cropping is not
  implemented.

Criterion 3 - [0.00/6.67 pts]:
- The output is a 2D image via mm.drawImg, not the resulting 1D signal
  on a single line with integer values separated by spaces, as
  required.

FINAL GRADE: 3.33/33.33

Comments for the student: The submitted code does not match what the
statement asks. The drawn question requests a 1D morphological erosion
on a one-dimensional signal; the code uses a 2D erosion function for
images. Re-read the assigned statement carefully before choosing the
implementation.
```

Q2 — syntax error prevents execution (high confidence, low grade due to a real failure): here the AI correctly identified the type (Otsu thresholding), supported by the official statement, but the code contained an indentation error that prevented execution — evidenced both by the static reading and by the absence of valid output in the VPL tests.

```
Identified type: TYPE A (Thresholding - manual T + Otsu)
Confidence: High - official statement confirms the type and variation

Criterion 1 - [6.67/10.00 pts]:
- Correctly reads the three parameters L, C, and T on separate lines
  and the input image. However, the definition of a helper thresholding
  function is incorrectly indented within the global scope, causing an
  indentation error that prevents the program from running completely.

Criterion 2 - [0.00/16.67 pts]:
- The fixed-threshold T binarization is correctly structured, but the
  Otsu threshold computation depends on the malformed helper function,
  which references a variable out of scope. The indentation error
  prevents any processing.

Criterion 3 - [0.00/6.67 pts]:
- Due to the indentation error, no output is produced.

FINAL GRADE: 6.67/33.33
```

```
Comments for the student: The overall structure is correct (reading L,
C, T, and the image), but there is an indentation error that prevents
execution. Review the Python syntax before submitting and test the code
locally before sending it to VPL.
```

Q3 — function mismatch, but functional equivalence confirmed by real evidence (high confidence, fair grade): the statement explicitly asked for the use of `mm.ero` (OpenCV-integrated version), but the code uses `mm.ero0` (manually implemented flat version). Since `execution.txt` showed all 10 VPL test cases passing, the AI did not penalize the function mismatch — it recognized the functional equivalence demonstrated by the real tests, but still noted the observation in the *feedback*.

```
Identified type: TYPE C1 (2D Erosion - flat, no weights)
Confidence: High - official statement confirms 2D erosion with
integrated OpenCV (mm.ero), and the code implements flat erosion via
mm.ero0

Criterion 1 - [10.00/10.00 pts]:
- Correctly reads height, width, the image matrix, and the dimensions
  and values of structuring element B.

Criterion 2 - [16.67/16.67 pts]:
- The statement asked for the use of mm.ero (integrated OpenCV), but
  the code uses mm.ero0 (flat, no weights). Despite the function
  mismatch, the operation was correctly implemented with the structuring
  element read: all 10 tests run on VPL passed completely (10/10),
  confirming functional equivalence for the tested cases.

Criterion 3 - [6.67/6.67 pts]:
- Prints the resulting image in the correct format; the tests confirmed
  the correct output.

FINAL GRADE: 33.34/33.33

Comments for the student: The code is correct and functional, with all
10 tests passing. Note: the statement explicitly asked for mm.ero
(OpenCV version), but you used mm.ero0 (flat version). Although the
results were equivalent in the tests, follow exactly the function
requested in the statement in future exams.
```

Note that, in this composite case, the AI assigned **23.7 fewer points** than VPL's objective grade, driven mainly by Q1. The aggregate difference alone would already be a signal for investigation (Section D.7), but the system itself hands the instructor the likely cause of each question: the `Revisar_Manualmente` column of the CSV is marked "YES" whenever at least one of the student's questions receives low confidence, with the specific reason recorded in the corresponding confidence column — reducing the time the instructor needs to spend finding, among a hundred students, the few cases that actually deserve manual attention before the official grade is assigned. The case of Q3, in turn, illustrates the opposite care: the divergence between statement and code did **not** result in undue penalization, because the real execution evidence confirmed the functional equivalence — exactly the behavior that Section D.3 describes for these two layers of evidence.

E.7 Sending *feedback* by e-mail

After the rubrics are generated, the `enviar_email.py` *script* sends each student an e-mail with `rubrica.txt` attached. The e-mail body is defined in `config.yaml`, under `templates.corpo`, and is interpolated with `{nome_pasta}` and `{login}`:

```yaml
email:
  smtp_server: smtp.ufabc.edu.br
  smtp_port: 587
  from_address: professor@ufabc.edu.br
  password: "YOUR_PASSWORD_HERE"     # never version real credentials
  use_tls: true

templates:
  assunto: "LLM-Generated Rubrics and Grading - Mock Exam - {login}@aluno.ufabc.edu.br"
  corpo: |
    Dear {nome_pasta},
    ...
```

! Credential security

`config.yaml` contains real secrets (SMTP password, API keys). It must be listed in the project's `.gitignore` and **never** be published, shared, or versioned — including in e-mail attachments, screenshots, or public repositories.

A real example of an e-mail sent to a student (data **anonymized**, with name and login replaced by a fictitious case) is reproduced below, to illustrate the didactic tone and the caveats made explicit to the student:

! Example of an individual message (anonymized)

Dear [Student Name] - [login],
Your grade for Mock Exam 4 is available on Moodle.
Below I'm sending the competencies assessed and a detailed review of your code, automatically generated by Artificial Intelligence (AI).
The "rubrica.txt" attachment contains the AI's full feedback (I recommend downloading it and opening it with a text editor or notepad).
Please note that this AI-generated review MAY contain inaccuracies or errors, but it serves as excellent support for your learning process in the course.
A pedagogical suggestion is to submit your code (contained in this file), together with the RUBRIC below, to other LLM models for comparison. This can help identify possible discrepancies, in addition to offering different perspectives on your code and on the assessment criteria.
If you have questions or notice any glaring inconsistency in the review shown on Moodle, I am available for clarification.
Best regards,
Prof. Francisco Zampirolli
PS.: Explaining the process: the latest version of your code saved on Moodle was attached to the prompt and sent to one of the LLMs chosen in an integrated way by our assessment system (models = ("deepseek-chat")). The process is repeated until a response with a valid grade is obtained (between 0 and 100).

Three didactic elements deserve attention in this text: (i) the explicit warning that the review **may contain errors**; (ii) the invitation for the student to **contrast the assessment with other LLMs**, turning the AI into an active study tool rather than a passive verdict; and (iii) the transparent explanation of the automated process, including the model(s) used and the reprocessing policy until a valid grade is obtained.

E.8 Risks, ethical limits, and instructor responsibility

! The instructor cannot simply adopt the AI's grade

This is the central point of this appendix: **under no circumstances should the grade suggested by the AI be automatically assigned to the student as the official grade**. Language models can hallucinate — award points for criteria not met, "invent" that a function was called correctly when it was not, or penalize correct code due to misinterpretation of the rubric. The final grade for any assessment activity must **always** result from the instructor's manual review (or from objective grading via test cases, as in VPL/MCTest), with AI serving, at most, as a **first pass** or as **support material for the student**, never as the decision-making authority.

Some practical precautions adopted in the use of **vpl-ai-feedback**, recommended to any instructor wishing to adapt the system:

- **Clear separation between the official grade and AI feedback.** In the case reported, the mock-exam bonus grade always came from VPL, never from the AI (Section D.1). The AI rubric was communicated to the student as supporting material, with an explicit warning that it may contain errors.
- **Transparency with the student.** The e-mail sent (Section D.6) makes explicit that the assessment was AI-generated, states which model(s) were used, and invites the student to compare with other LLMs — reducing information asymmetry and encouraging critical thinking about the review they received.
- **Reprocessing instead of an invalid *cache*.** The system only writes `rubrica.txt` when it obtains a valid assessment (Section D.4); this prevents a malformed, incomplete, or visibly inconsistent AI response from being presented to the student as if it were final.
- **Monitoring discrepancies.** The consolidated CSV report (the `Diferenca = Total_IA - Total_Moodle` column) allows the instructor to quickly identify the cases where AI and objective grading disagree the most — as in the example in Section D.5 (+20 points) —, prioritizing those cases for manual review.
- **Open channel for disputes.** The final e-mail explicitly offers the student the option to report inconsistencies, keeping the instructor as the final authority over the grade.
- **Personal data kept out of the *prompt*.** As described in Section D.2, name, e-mail, login, student ID, and national ID are never sent to the LLM API, reducing privacy risks when using external AI providers.

E.9 Final remarks

vpl-ai-feedback shows how AI can enrich the feedback cycle in programming courses with a high volume of submissions, offering each student a qualitative, individualized reading of their own code — something unfeasible to do manually for a hundred students, across multiple questions and exams throughout the term. The opinion survey cited in Section D.1 suggests that this type of feedback is well received by students. Even so, responsible use of the system depends on a non-negotiable principle, reiterated throughout this appendix: **AI complements, but does not replace, the instructor's assessment**, and the official grade for any assessment activity must continue to be defined by objective grading (VPL/MCTest, Appendices A and B) and/or human judgment — never by the raw grade returned by a language model.

THE JOURNEY OF DIP & CV

SYSTEMATIZING KNOWLEDGE, FROM PIXEL TO INTELLIGENCE

Part I — Foundations of DIP

- 1. Fundamentals and First Steps
- 2. Sampling, Quantization, and Connectivity
- 3. Spatial Operations and Filtering
- 4. Mathematical Morphology and Segmentation
- 5. Transforms and Compression

Part II — Computer Vision

- 6. Document Analysis and Inspection
- 7. Classification and Pattern Recognition
- 8. Scene Understanding and Segmentation
- 9. Deep Learning (Deep Learning)

Essential Appendices

- **A:** MCTest
- **B:** Moodle/VPL
- **C:** SEB/Simulators
- **D:** Gemini/Teaching Material
- **E:** AI in the vpl-ai-feedback
- **F:** Perception and Assessment

INPUT IMAGE (PIXEL)

DIP (PROCESSING)

CV (VISION)

INTELLIGENT OUTPUT (SEMANTIC)

MASTER THE PIXELS, BUILD THE FUTURE.

From simple histograms to complex scene understanding with deep learning, this work offers a complete roadmap. Transform images into data, and data into intelligent decisions.

Your ability to shape computational perception starts here. Stay ahead, explore, and innovate!

Available in digital format (HTML, PDF, and Jupyter Notebook), with simulators and exercises, including test cases with VPL and Moodle.

https://fzampirolli.github.io/dip-cv/

SCAN AND ACCESS EXTRA CONTENT