PDI&VC

Processamento Digital de Imagens e Visão Computacional

Fundamentos, Algoritmos e Aplicações Integradas

AUTOR: FRANCISCO DE ASSIS ZAMPIROLLI

Processamento Digital de Imagens e Visão Computacional

Livro interativo com C++

Francisco de Assis Zampirolli

Universidade Federal do ABC

20 de setembro de 2026

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

CATALOGAÇÃO NA FONTE
SISTEMA DE BIBLIOTECAS DA UNIVERSIDADE FEDERAL DO ABC

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

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

ISBN: [ISBN A DEFINIR] (1ª Edição)

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

CDD [XX] ed. – [CÓDIGO CDD]

Sumário

Lista de Figuras

Lista de Tabelas

PDI e VC — C++

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

Organização

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

---

Prefácio

Este livro constitui uma **obra em permanente atualização** nas áreas de **Processamento Digital de Imagens (PDI)** e **Visão Computacional (VC)**, concebida como material didático interativo para cursos de graduação e pós-graduação em Computação, Engenharia e áreas afins.

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

Este não é um livro estático. Seu conteúdo evolui continuamente à medida que exemplos são aprimorados, novas seções são incorporadas e as abordagens pedagógicas são refinadas com base na experiência de uso e no retorno de estudantes e docentes. Dessa forma, a obra é tratada como um *projeto em constante evolução*, buscando acompanhar tanto os avanços tecnológicos quanto as melhores práticas de ensino em PDI e VC.

Contexto da primeira edição

O conteúdo desta edição foi elaborado entre maio e agosto de 2026, durante a primeira oferta da disciplina de Processamento Digital de Imagens baseada neste material didático. A disciplina foi ministrada em duas turmas de graduação — majoritariamente compostas por estudantes de Ciência da Computação, no período matutino — e em uma turma de pós-graduação em Ciência da Computação, todas na UFABC. Esta primeira edição representa o resultado dessa oferta inicial, consolidando o conteúdo desenvolvido, testado e continuamente aperfeiçoado ao longo do período letivo.

A metodologia de ensino adotada seguiu uma sequência estruturada em cada aula. Inicialmente, era exibido um vídeo curto, com duração média de 7 minutos, gerado no **Gemini Notebook**, alternando entre a apresentação da parte conceitual do capítulo e a resolução dos Exercícios de Programação (EPs). Neste segundo caso, quase todos os EPs eram resolvidos durante a própria aula. Em seguida, era apresentado um conjunto de aproximadamente 12 *slides*, também gerados no **Gemini Notebook**, tendo como fontes o PDF completo do livro e o arquivo `morph.py`. Esses *slides* eram produzidos a partir de um *prompt* específico para a geração do conteúdo teórico e prático de cada capítulo.

Após essa etapa inicial, restavam aproximadamente 100 minutos de aula para a exploração dos dois *notebooks* Colab do capítulo, um teórico e outro prático, com ênfase nos simuladores interativos e na execução dos blocos de código, permitindo aos estudantes modificar parâmetros e observar seus efeitos. Nas aulas realizadas em laboratório, os estudantes transferiam as respostas dos EPs desenvolvidas no Colab diretamente para as atividades VPL do Moodle, nas quais eram submetidas à avaliação automática. Esse processo de publicação e utilização dos EPs é apresentado ao final deste prefácio.

A avaliação contínua incluiu simulados quinzenais, aplicados com o **SEB (*Safe Exam Browser*)** no Moodle, contendo EPs semelhantes aos das listas de exercícios. Cada simulado concedia um bônus de 5% sobre a nota final. As duas provas da disciplina também utilizaram o SEB, mas foram compostas por questões parametrizadas geradas no **MCTest**, cuja descrição combina LaTeX e parâmetros no formato `[[code:variavel]]`, definidos em trechos de Python delimitados por `[[def: ... ]]` na própria questão. Esse processo permitiu gerar 110 variações de prova, sorteadas individualmente entre os estudantes.

O Apêndice A detalha a criação dessas questões no MCTest e sua exportação para o Moodle; o Apêndice B descreve a configuração das atividades VPL, incluindo o processo de publicação dos EPs; o Apêndice C

apresenta a configuração do SEB para os simulados e as provas; o Apêndice D descreve a geração dos vídeos e *slides* utilizados nas aulas com o **Gemini Notebook**; e o Apêndice E detalha o uso de **Inteligência Artificial (IA)** na geração de *feedback* socrático para atividades formativas e avaliativas, complementando a correção automática realizada pelo VPL.

Como este livro é produzido

O conteúdo deste livro é desenvolvido em **Quarto** e armazenado na pasta `all` do repositório **github.com/fzampirolli/pdi-vc**. A partir de um único código-fonte, o livro é automaticamente gerado e publicado em diferentes formatos, apresentados na Tabela 1.

Embora a edição em PDF registre o estado da obra ao término da primeira oferta da disciplina em 2026, o desenvolvimento do livro continua de forma permanente. A cada atualização do repositório GitHub, são geradas automaticamente novas versões das páginas HTML, do PDF e dos notebooks, disponibilizando imediatamente as melhorias aos leitores.

Tabela 1: Diferentes formatos de publicação gerados automaticamente a partir do mesmo código-fonte.

Formato	Descrição
HTML	Versão para *web*, com simuladores interativos e navegação entre capítulos: fzampirolli.github.io/pdi-vc/
PDF	Versão adequada para impressão ou leitura *offline*: livro.pt.py.pdf
Notebooks (`.ipynb`)	Compatíveis com **Jupyter** e **Google Colab**, permitindo executar, modificar e experimentar os exemplos de código e os Exercícios de Programação (EPs). Um filtro personalizado mantém referências cruzadas, numeração de figuras e tabelas, além de formatar automaticamente as citações conforme o padrão ABNT.

A obra é publicada em **dez combinações** — cada trilha de código (**Python**, 9 capítulos; **C++**, capítulos 1 a 5) em cinco idiomas (Português, Inglês, Francês, Espanhol e Italiano). Com o cache de tradução já povoado, uma regeneração completa (tradução, reexecução dos *notebooks*, renderização de HTML e PDF e publicação) leva cerca de **72 minutos**, com paralelismo real de ~5 processos em média (pico de 8); a **primeira** geração de um idioma novo, com o cache vazio, é bem mais lenta — cerca de **80 minutos** por combinação, como observado nas primeiras gerações de espanhol e italiano. A Tabela 2 resume cada combinação (já com o cache povoado): número de páginas do PDF e tempo de renderização.

Tabela 2: Páginas do PDF e tempo de renderização por combinação (paralelismo real médio de ~5 processos, pico de 8). A combinação-base em Python/Português apenas renderiza as saídas já registradas nos *notebooks*-fonte; as demais reexecutam todo o código.

Combinação	Trilha	Idioma	Páginas	Tempo de render.
`py.pt`	Python	Português (base)	654	~7 min
`py.en`	Python	Inglês	644	~31 min
`py.fr`	Python	Francês	666	~31 min
`py.es`	Python	Espanhol	666	~31 min
`py.it`	Python	Italiano	658	~31 min
`cpp.pt`	C++	Português	434	~26 min
`cpp.en`	C++	Inglês	426	~34 min
`cpp.fr`	C++	Francês	434	~38 min
`cpp.es`	C++	Espanhol	430	~37 min
`cpp.it`	C++	Italiano	428	~35 min

Estado de validação. Apenas a combinação `py.pt` (Python, Português) é a fonte curada: é nela que o autor escreve, revisa e valida todo o conteúdo. As outras nove combinações — as trilhas em **C++** e as

traduções para inglês, francês, espanhol e italiano — são **geradas automaticamente**, por tradução via modelo de linguagem e transpilação de código, e ainda **carecem de revisão detalhada**. A atenção maior recai sobre os **textos embutidos em algumas figuras**: onde a versão traduzida ainda não foi produzida, a imagem aparece com o texto em português (cada figura traduzida segue o mesmo sufixo de idioma dos demais arquivos gerados, por exemplo `imagem_en.png`).

As páginas HTML, o PDF e os *notebooks* disponibilizados no projeto refletem sempre o estado mais recente do desenvolvimento. Quando uma **edição oficial** é publicada, ela passa a ser identificada por uma *tag* no repositório GitHub, permitindo reproduzir exatamente o conteúdo correspondente àquela edição. Assim, o leitor pode acompanhar tanto a evolução contínua do projeto quanto recuperar qualquer edição oficial anteriormente publicada.

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

Cada capítulo disponibiliza dois botões **Executar Colab** (Figura 1), que direcionam para ambientes complementares:

1. **Parte Teórica**, localizada na abertura de cada capítulo, contendo os conceitos apresentados e exemplos de código executáveis;
2. **Parte Prática**, na seção **Parte Prática com Exercícios de Programação** (EPs), dedicada aos exercícios e aos seus simuladores interativos.

Figura 1: Botão clicável **Executar Colab**, disponível no início das partes **Teórica** e **Prática** de cada capítulo, permitindo a execução interativa dos exemplos e exercícios. Na versão PDF, existe um *link* HTML direto entre a imagem do simulador e a sua legenda.

A geração dos *notebooks* é totalmente automatizada pelo *script* `gerar_notebooks_alunos.py`, que adapta o conteúdo para ambientes interativos, permitindo sua execução sem a necessidade de instalação do Quarto.

Uso de ferramentas de Inteligência Artificial

A concepção, o projeto pedagógico, a estrutura conceitual e o conteúdo fundamental deste livro são de **autoria exclusiva do autor**.

No processo de editoração e apoio ao desenvolvimento, foram utilizadas, em suas versões gratuitas, ferramentas de **Inteligência Artificial (IA)**, como **ChatGPT**, **Claude**, **DeepSeek** e **Gemini**, empregadas estritamente como recursos auxiliares. Seu uso restringiu-se ao apoio à revisão e ao aprimoramento do estilo textual, à otimização da sintaxe de códigos e à geração assistida de ilustrações conceituais e exemplos.

Todas as respostas e conteúdos sugeridos por essas ferramentas foram submetidos à **análise crítica, verificação, validação técnica, adaptação e integração pelo autor**, que assume a responsabilidade pelo texto, pelos códigos e pelos recursos pedagógicos apresentados. As ferramentas de IA **não são consideradas autoras ou coautoras da obra**, nem responsáveis pelas decisões intelectuais, técnicas ou pedagógicas que fundamentam seu conteúdo.

Destaca-se, ainda, que os **simuladores** e recursos interativos presentes no livro — disponibilizados nas versões HTML e Jupyter Notebook (IPYNB) — foram projetados e implementados como parte da abordagem pedagógica da obra, com o objetivo de proporcionar experimentação direta dos conceitos apresentados e favorecer a autonomia de aprendizagem do leitor.

Distinto do uso editorial descrito acima, o *pipeline* de geração multi-idioma (ver “Como este livro é produzido”) emprega um modelo de linguagem como **componente de engenharia do sistema**: a tradução automática do conteúdo entre português e outros idiomas, com cache incremental que evita retraduzir trechos inalterados.

Essa etapa usa a API paga do **DeepSeek** (modelo `deepseek-v4-flash`); o livro completo já foi traduzido do português para o **inglês**, o **francês**, o **espanhol** e o **italiano**, e os **Capítulos 1 a 5** contam ainda com uma trilha em **C++** que compila e executa de verdade — a um custo acumulado da ordem de poucos dólares, graças ao cache incremental.

A biblioteca `morph.hpp`

A biblioteca **`morph.hpp`** (Zampirolli et al., 2025) acompanha toda a obra como uma ferramenta de apoio ao ensino. Seu objetivo não é substituir bibliotecas consolidadas, como **OpenCV**, nem competir com elas em desempenho ou abrangência. Em vez disso, procura **tornar transparentes os algoritmos fundamentais de Processamento Digital de Imagens e Visão Computacional (PDI-VC)**, permitindo que o estudante compreenda sua implementação, modifique o código e desenvolva novas funcionalidades.

É o equivalente em C++ da `morph.py`: *header-only* (basta um `#include "morph.hpp"`), com os **mesmos nomes de função** no *namespace* `mm::` — quem já conhece `mm.dil()`, `mm.dil0()` ou `mm.gradm()` em Python reconhece de imediato `mm::dil()`, `mm::dil0()` e `mm::gradm()` em C++. Divergência de nome entre as duas versões é considerada um defeito da biblioteca, não uma escolha de projeto.

> **i** Herança Técnica
>
> A estrutura da `morph.hpp` (assim como a de sua contraparte `morph.py`) tem como base ferramentas de Morfologia Matemática desenvolvidas no Brasil, como **MMachLib** (Lotufo et al., 1997) e **MMach** (Barrera et al., 1998), além da **mmorph**, utilizada na obra de Dougherty; Lotufo (2003). Essas bibliotecas serviram de referência para o ensino da área durante décadas.

Essa filosofia pode ser observada, por exemplo, nos operadores morfológicos de dilatação:

- `mm::dil0`: implementação didática da dilatação para **elementos estruturantes planares**, seguindo diretamente a definição clássica da Morfologia Matemática;
- `mm::dil1`: implementação didática da dilatação para **funções estruturantes (*kernels* não planares)**, seguindo rigorosamente sua formulação matemática;
- `mm::dil`: por padrão, delega para `mm::dil0()` — a versão didática planar, numericamente equivalente a `cv::dilate` para elementos estruturantes desse tipo. A implementação otimizada, baseada em **OpenCV** (`cv::dilate`), só entra em ação se o programa for compilado com a *flag* `-DMM_USE_OPENCV`.

> **!** Uma diferença deliberada em relação à `morph.py`
>
> Em Python, `mm.dil()` (sem sufixo) **já é** a implementação otimizada por padrão. Em C++, `mm::dil()` (mesmo nome, mesma assinatura) só se torna a versão otimizada se o OpenCV for explicitamente linkado na compilação; sem isso — que é justamente o caso padrão, inclusive no Moodle/VPL — `mm::dil()` se comporta como `mm::dil0()`. A escolha evita que a trilha C++ dependa do OpenCV apenas para compilar e rodar um exercício simples: `g++ arquivo.cpp -o arquivo` já é suficiente. Quem quiser a versão acelerada localmente compila com `g++ -DMM_USE_OPENCV arquivo.cpp -o arquivo $(pkg-config --cflags --libs opencv4)`.

A biblioteca também oferece funções para leitura, exibição, conversão de cores, filtragem e morfologia matemática por meio de uma interface simples:

```cpp
#include "morph.hpp"

int main() {
    mm::Image img = mm::gray(mm::read("lena.jpg"));  // leitura e conversão para níveis de
cinza
    mm::Image grad = mm::gradm(img);                 // gradiente morfológico
    mm::show(grad, "saida.png");                     // exibição (grava em arquivo)
```

```
}
```

Ao contrário da versão Python, `mm::show()` exige um caminho de saída explícito: cada célula de código C++ do livro roda como um processo isolado (compilado e executado via `%%writefile` + `!g++ ...` no Colab), sem o contador global de figuras que só faz sentido dentro de um único processo Jupyter.

Além disso, todos os projetos do **Gemini Notebook** da trilha C++ incluem o arquivo `morph.hpp`, permitindo consultar e discutir diretamente a implementação dos algoritmos apresentados no livro. Dessa forma, o estudante pode utilizar o próprio Gemini Notebook como um assistente de estudos, fazendo perguntas como:

> Explique como foi implementada a função `mm::dil0` do `morph.hpp`, mostrando o código e comentando cada etapa de forma didática. Além disso, explique a diferença entre `mm::dil0()` e `mm::dil()` sem a *flag* `-DMM_USE_OPENCV`.

! Implementações didáticas e implementações otimizadas

Em vários capítulos, um mesmo algoritmo é apresentado em duas versões. As funções com sufixo `0`, `1`, etc. (por exemplo, `mm::dil0()` e `mm::dil1()`) são implementações didáticas, desenvolvidas para facilitar o entendimento dos algoritmos e disponíveis independentemente da flag de compilação. Já as funções sem sufixo (como `mm::dil()`) usam a implementação otimizada baseada em OpenCV **somente** quando compiladas com `-DMM_USE_OPENCV`; caso contrário, comportam-se como a versão didática correspondente.
Nas atividades avaliadas pelo **VPL do Moodle**, a compilação segue o padrão sem `-DMM_USE_OPENCV` — não por limitação de memória (como ocorre com scikit-learn/scikit-image na trilha Python), mas para que nenhum EP em C++ dependa de o OpenCV estar instalado no ambiente de execução. Na prática, isso significa que, no VPL, `mm::dil()` e `mm::dil0()` produzem exatamente o mesmo resultado. Localmente — por exemplo, no **VS Code** ou no **Google Colab** —, o estudante pode optar por compilar com `-DMM_USE_OPENCV` para comparar a versão otimizada.

O código-fonte da biblioteca está disponível em:

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

Exercícios de Programação (EPs) e Validação Automática

Cada unidade do livro inclui **Exercícios de Programação (EPs)** práticos e de complexidade crescente, projetados para consolidar os conceitos apresentados ao longo do capítulo. Cada EP é acompanhado por um **simulador interativo**, disponível nas versões HTML e IPYNB, que permite ao estudante manipular parâmetros, visualizar o comportamento dos algoritmos e desenvolver uma compreensão intuitiva do problema antes de iniciar sua implementação. Dessa forma, o processo de aprendizagem combina experimentação, programação e validação automática.

A validação das soluções é feita localmente pela classe `TestSuite` (`testsuite.py`), que compara a saída do programa com os arquivos de casos de teste (`.cases`):

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

O sistema suporta múltiplas linguagens (Python, Java, C++, C, JavaScript e R) e pode ser integrado diretamente ao Moodle. Para docentes interessados no passo a passo completo de publicação dos EPs como atividades VPL, consulte o **Guia do Professor**, ao final deste prefácio, e o Apêndice B.

Código aberto

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

Cada versão do livro é arquivada permanentemente no **Zenodo** com um *DOI citable*. Para citar este material em trabalhos acadêmicos:

> ZAMPIROLLI, Francisco de Assis. **PDI+VC — Processamento Digital de Imagens e Visão Computacional**. UFABC, 2026. DOI: 10.5281/zenodo.20784605

Cabe registrar, por oportuno, que o conteúdo exposto nesta obra reflete o olhar crítico, a proposta pedagógica e a experiência docente de seu autor. Assim, as análises, abordagens e opiniões expressas ao longo deste livro representam tão somente o entendimento de seu idealizador, não constituindo nem refletindo um posicionamento oficial ou institucional da Universidade Federal do ABC (UFABC).

Antes de começar: *Notebooks* em Python

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

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

- **Execução:** células de código são identificadas por `[ ]`. A execução pode ser realizada com `Shift + Enter` ou pelo botão ▷ da interface.
- **Ambientes:** os *notebooks* podem ser executados localmente, por meio do Jupyter ou do Visual Studio Code (VS Code), bem como em ambientes de nuvem, como o Google Colab.

> Nota sobre o formato
>
> Em ambientes interativos, o código pode ser modificado e executado. Nas versões estáticas (HTML ou PDF), os blocos de código têm finalidade de leitura e referência, sem prejuízo à integridade das explicações.

Guia do Professor: Publicação de EPs no Moodle

Esta seção destina-se a docentes que desejam integrar os Exercícios de Programação (EPs) às atividades ***VPL (Virtual Programming Lab)*** *do Moodle, complementando o Apêndice B.*

Integração com o Moodle (VPL)

Os EPs dos *notebooks* podem ser convertidos em atividades **VPL** no Moodle. O *script* `ep_tools.py` (executado via `make build`) automatiza a exportação dos EPs do final de cada capítulo em duas etapas:

1. **Extração** (`make eps`): gera um HTML interativo independente por EP, com enunciado, exemplos e simulador, em `gen/book/eps/<versao>/`. Veja a lista completa em: https://fzampirolli.github.io/pdi-vc/eps/py.pt/index.html
2. **Conversão** (`make moodle`): transforma o HTML em um fragmento autocontido, sem dependência de CSS externo, pronto para ser colado no editor do Moodle, em `gen/book/eps/<versao>_moodle/`. Veja um exemplo em: https://fzampirolli.github.io/pdi-vc/eps/py.pt_moodle/EP01_01.html

Adicionalmente, todos os simuladores interativos desenvolvidos ao longo do livro podem ser acessados diretamente em https://fzampirolli.github.io/pdi-vc/simuladores/py.pt/.

Outras trilhas e idiomas

Os *links* acima usam `py.pt` como exemplo. Para acessar EPs ou simuladores de outra combinação linguagem/idioma, basta trocar esse trecho na URL — por exemplo, `cpp.it` para a trilha C++ em italiano: `https://fzampirolli.github.io/pdi-vc/eps/cpp.it/index.html`. A estrutura é a mesma para todas as combinações disponíveis, em `/eps/<versao>/`, `/eps/<versao>_moodle/` e `/simuladores/<versao>/`.

Ao publicar com `make publish`, essas **duas versões** de cada EP ficam disponíveis nos *links* indicados. O professor pode combinar as duas estratégias: colar a versão Moodle no editor VPL (com simulador funcional) e incluir no enunciado um *link* para a versão completa, onde as fórmulas são renderizadas corretamente pelo MathJax.

Revisão obrigatória antes de publicar no Moodle

Embora automatizada, a conversão exige revisão do professor em razão das limitações do editor TinyMCE/HTML Purifier do Moodle:

- **Fórmulas matemáticas** — O TinyMCE descarta barras invertidas (\), corrompendo equações LaTeX. Por isso, a versão Moodle converte fórmulas simples para HTML puro (entidades como `&ge;`, `&times;`). Fórmulas complexas (`\begin{cases}`, matrizes, integrais) podem sair incompletas — revise e, se necessário, reescreva-as em texto ou indique a versão completa via *link*.
- **Figuras externas** — Imagens complementares devem ser inseridas manualmente na atividade VPL.
- **Simuladores** — Funcionam perfeitamente desde que utilizem JavaScript padrão com estilos *inline* e manipulação direta do DOM (`getElementById`). **Atenção:** se incluir fórmulas em LaTeX que contenham o caractere \ no Moodle, os simuladores podem parar de funcionar.

Como Publicar no Moodle

1. Acesse a versão Moodle do EP desejado:

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

2. Copie o código-fonte completo da página (`Ctrl+U` → `Ctrl+A` → `Ctrl+C`).
3. No Moodle, crie a atividade VPL, acesse o editor HTML, cole o conteúdo (`Ctrl+V`) e salve.
4. Importe os arquivos `.cases` da pasta `all/capXX/casos/` nas configurações de testes do VPL.

Reaproveitamento dos Casos de Teste

Os arquivos `.cases` são compatíveis com o VPL, permitindo usar o mesmo conjunto de testes tanto no desenvolvimento local quanto na correção automatizada no Moodle.

Parte I

Parte I — Fundamentos de PDI

Capítulo 1

Fundamentos e Primeiros Passos

Este capítulo inaugura a **Parte 1** do livro, dedicada aos fundamentos do Processamento Digital de Imagens (PDI). São apresentados a representação matemática de imagens digitais e os principais métodos para sua manipulação.

Utiliza-se a linguagem C++ e a biblioteca `morph.hpp`, uma porta didática da `morph.py` (ZAMPIROLLI, 2025).

1.1 Objetivos

Ao final deste capítulo, você será capaz de:

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

1.2 Antes de começar: *Notebooks* interativos

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

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

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

1.3 Fundamentos

O estudo de sistemas baseados em imagens compreende um ecossistema de disciplinas integradas que transformam dados visuais brutos em conhecimento estruturado. Enquanto algumas áreas focam na geração

de representações, outras dedicam-se ao tratamento e à análise desses dados para dar suporte a aplicações tecnológicas complexas.

O diagrama apresentado no Figura 1.1 estabelece a distinção e complementaridade entre o Processamento Digital de Imagens (PDI) e a Visão Computacional (VC). O PDI, destacado em **verde**, tem como foco a transformação de imagem para imagem, visando melhoria de qualidade ou pré-processamento, como a remoção de ruídos e realce de contraste.

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

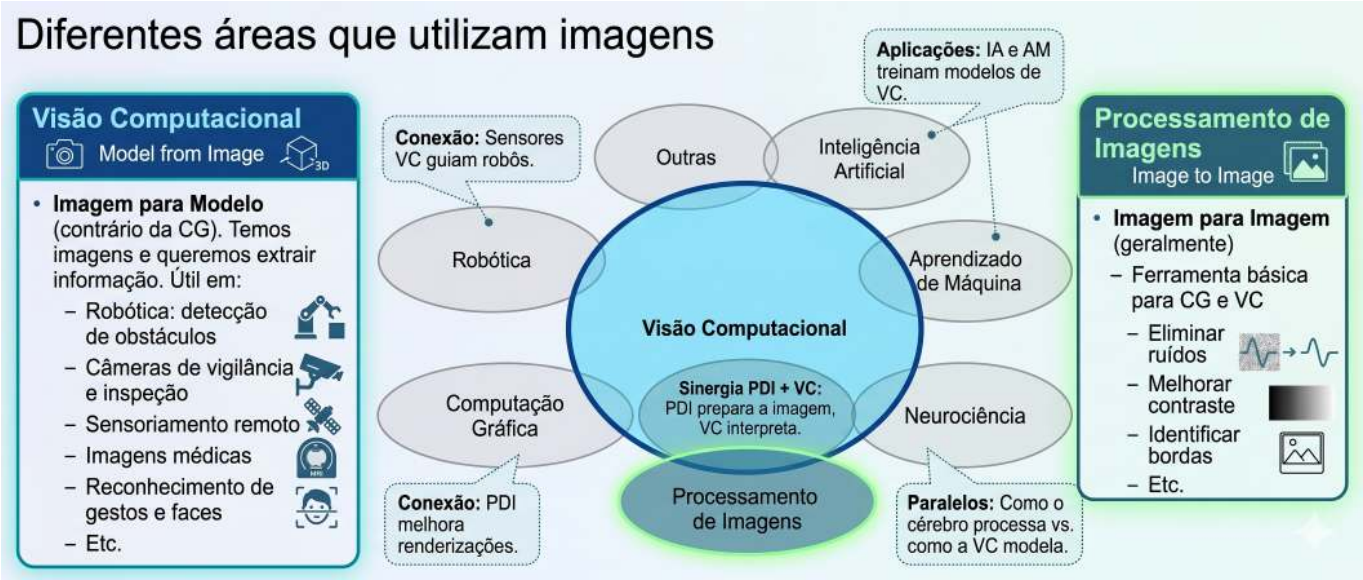


Figura 1.1: Diagrama relacional detalhando as distinções fundamentais, sinergias e interconexões entre PDI e VC no contexto de sistemas baseados em imagens.

1.3.1 👁 Visão Computacional

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

1.3.2 🖼 Processamento Digital de Imagens (PDI)

- **Foco:** *Imagem* → *Imagem* (geralmente - transformação de uma imagem em outra).
- **Objetivo:** Melhorar a qualidade visual ou extrair características de baixo nível.

- **Aplicações comuns:**
  - Eliminação de ruídos (filtros de média, mediana, gaussiano).
  - Melhoria de contraste (equalização de histograma, ajuste gamma).
  - Detecção de bordas (Sobel, Canny, Laplaciano).
  - Segmentação (limiarização, crescimento de regiões, *watershed*).
  - Transformações geométricas (redimensionamento, rotação, correção de perspectiva).
- **Relação com outras áreas (ver Tabela 1.1):**
  - É a **base** para a maioria dos sistemas de VC (pré-processamento).
  - A Computação Gráfica frequentemente aplica PDI para pós-processamento (ex.: suavização, realce).
  - Técnicas de IA podem otimizar parâmetros de processamento (ex.: aprendizado de filtros).

Tabela 1.1: Conexão entre PDI, VC e outras áreas da ciência.

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

1.4 Etapas do PDI

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

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

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

1.5 Formação da Imagem e o Espectro

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

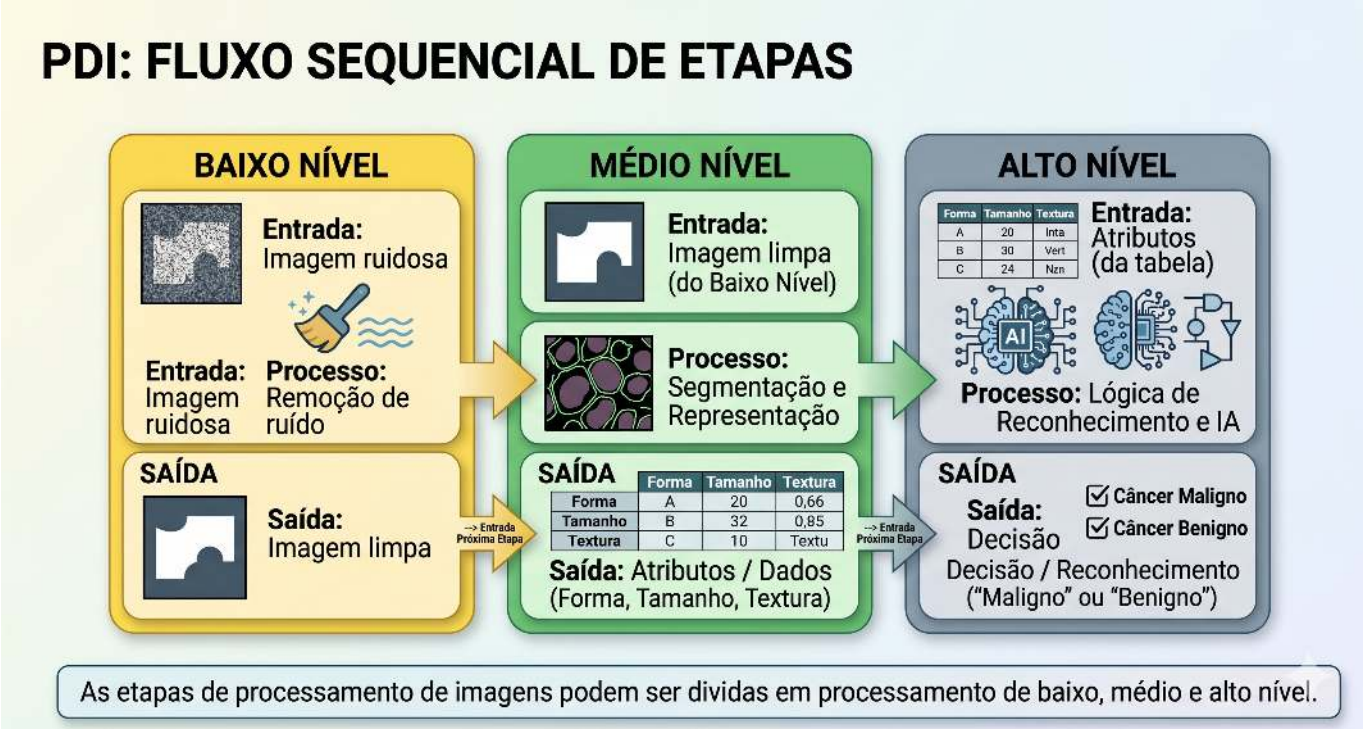


Figura 1.2: Representação do fluxo sequencial de processamento: a saída de cada nível torna-se a entrada do nível subsequente.

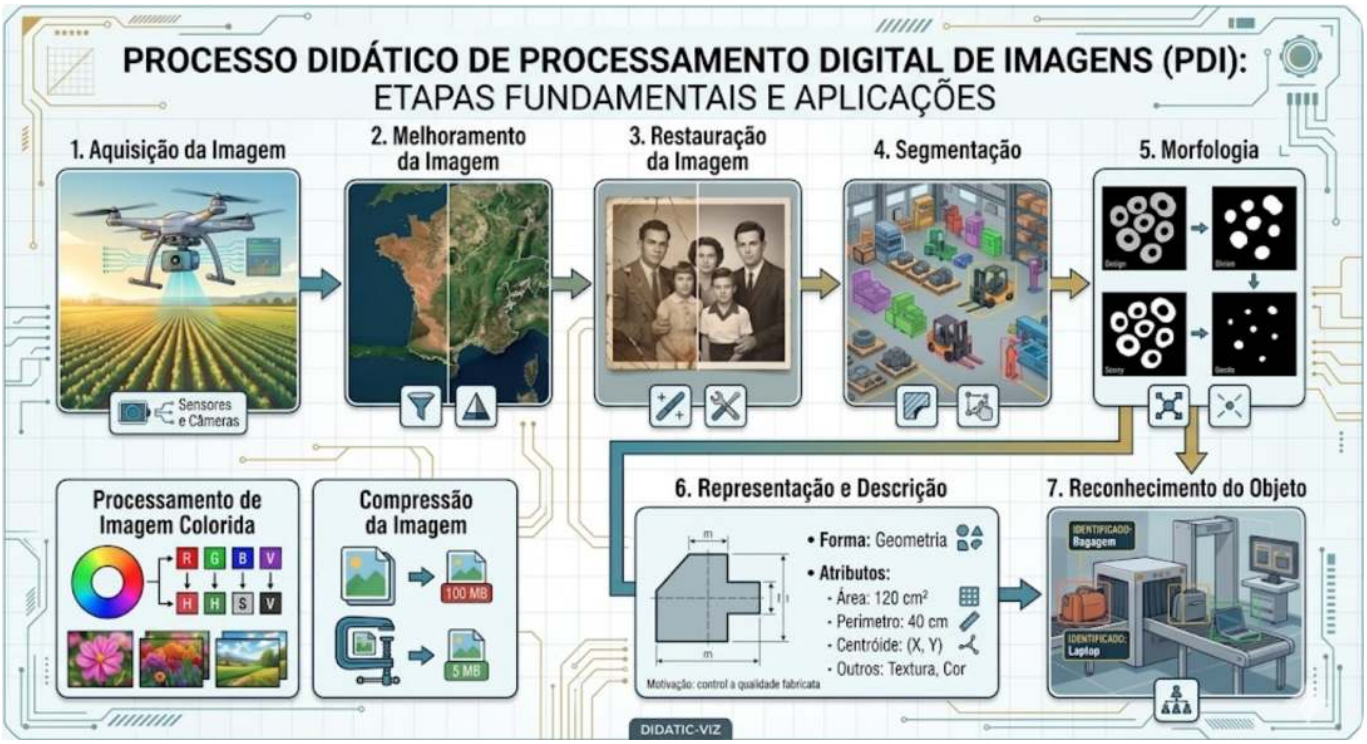


Figura 1.3: Fluxo detalhado do PDI: da aquisição sensorial à extração de atributos e reconhecimento automatizado, incluindo em processamento colorido e compressão.

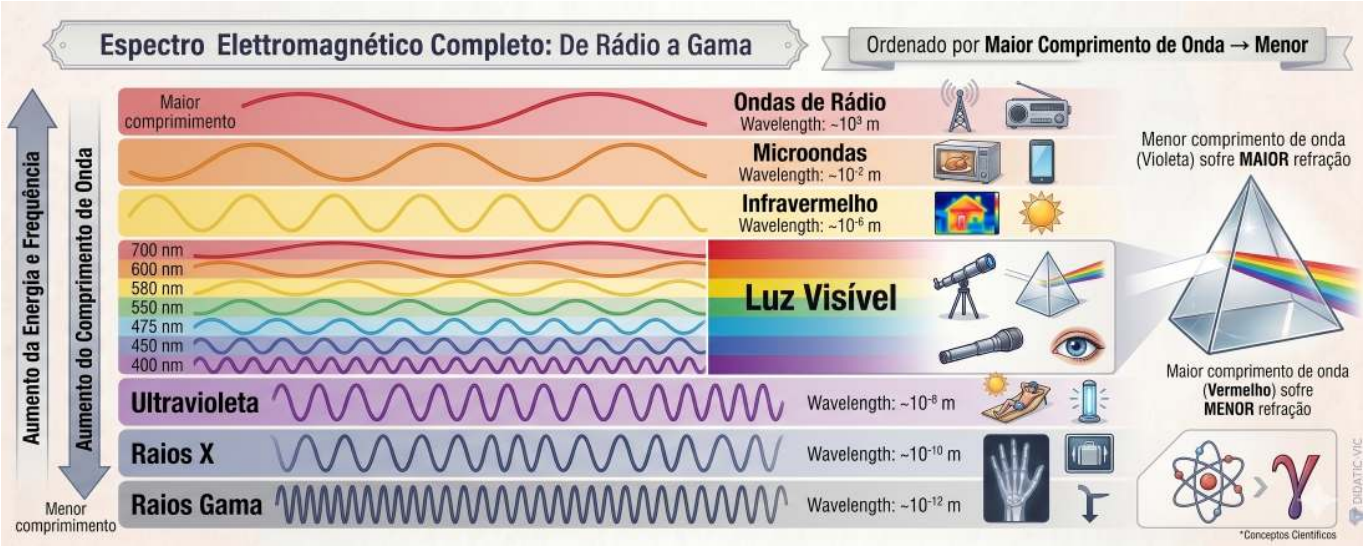


Figura 1.4: Representação do espectro visível e sua posição em relação às demais radiações eletromagnéticas, destacando a variação dos comprimentos de onda de 380 nm a 750 nm.

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

A partir desse processo físico de aquisição, torna-se possível modelar matematicamente a imagem digital como uma função bidimensional discreta, na qual cada ponto da cena é representado por amostras numéricas de intensidade luminosa, formalizando assim os conceitos de pixel e de imagem digital apresentados na próxima seção.

1.6 O que é uma Imagem Digital?

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

O que é um Pixel?

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

Representação Matemática

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

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

Onde:

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

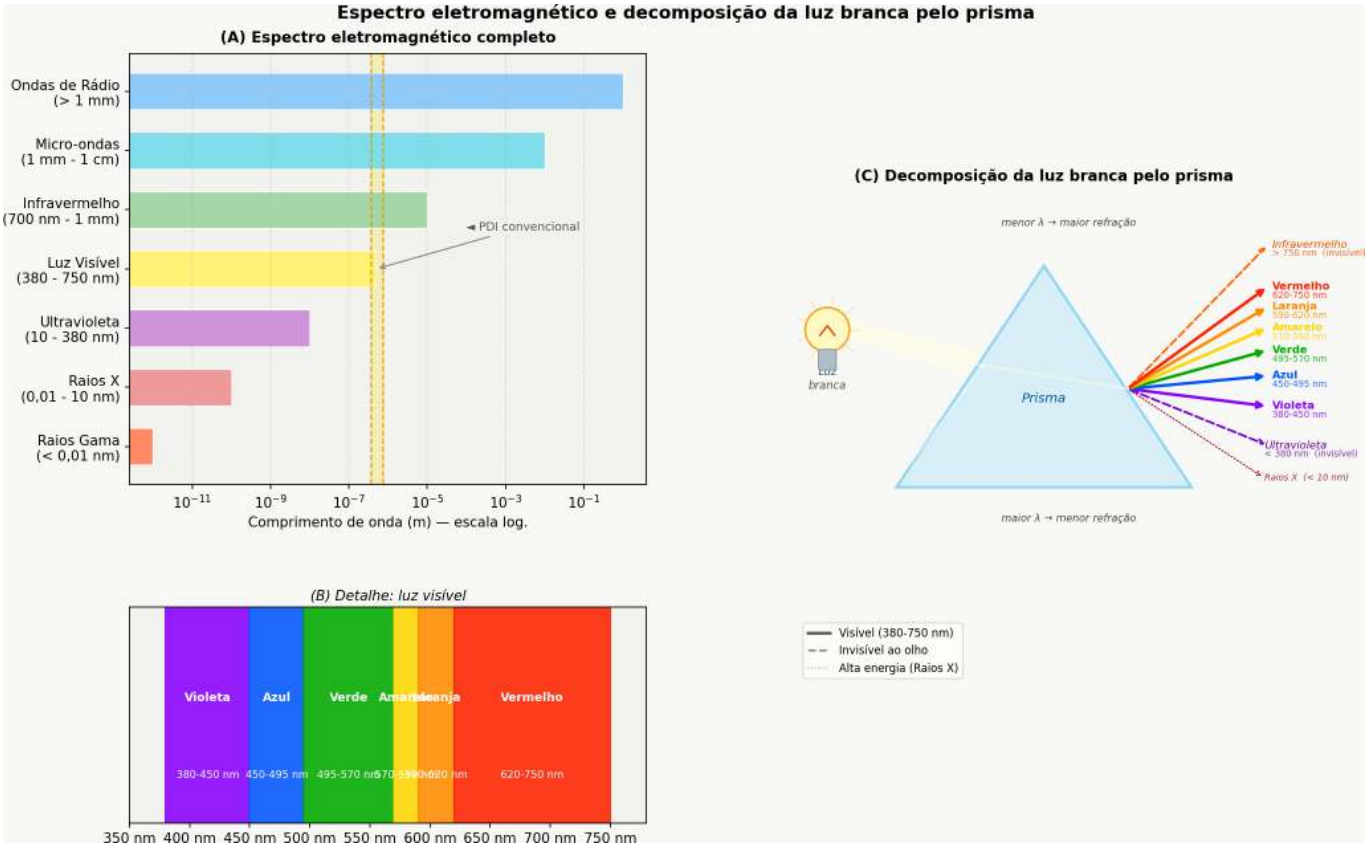


Figura 1.5: (A) Espectro eletromagnético completo em escala logarítmica, com destaque para a faixa visível. (B) Detalhe da luz visível (380-750 nm) e suas cores. (C) Decomposição da luz branca pelo prisma: menor comprimento de onda sofre maior refração, separando UV, visível e infravermelho.

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

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

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

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

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

Exemplo prático: Uma imagem colorida no modelo RGB pode ser representada matematicamente por uma função que associa três valores de intensidade a cada pixel. Computacionalmente, essa representação corresponde a três matrizes sobrepostas — os canais vermelho (*Red*), verde (*Green*) e azul (*Blue*) — nas quais cada elemento armazena a intensidade luminosa do respectivo canal em uma determinada posição da imagem.

Para viabilizar os experimentos práticos de PDI e VC, este livro utiliza C++17 e um conjunto mínimo de bibliotecas voltadas à manipulação matricial, ao processamento de imagens e à visualização de resultados, apresentadas a seguir.

1.7 Configuração do Ambiente

Este material utiliza **C++17** e a biblioteca de cabeçalho único **morph.hpp**, originalmente proposta para Python em Zampirolli (2025). Aqui, é utilizada uma versão mínima e didática da `morph.py`, adaptada para compilação simples com `g++`, conforme apresentado na Tabela 1.3.

Cada célula de código é compilada e executada como um processo isolado (`%%writefile arquivo.cpp` seguido de `g++`). Diferentemente do *kernel* Python, não há estado persistente entre as células: as imagens produzidas por uma célula são salvas em arquivos para serem lidas pela próxima.

Os **Exercícios de Programação (EPs)** apresentados ao final dos capítulos podem ser validados pelo mesmo módulo `testsuite.py` utilizado na trilha Python, que compila a solução com `g++` e compara sua saída com os casos de teste. Assim, os mesmos casos de teste (`.cases`) podem validar soluções em C++, Python e outras linguagens, tanto nos *notebooks* quanto no **Moodle/VPL**.

Tabela 1.3: Principais bibliotecas e ferramentas utilizadas na trilha C++ deste livro.

Biblioteca / Ferramenta	Função principal
`g++` (C++17)	Compilação de cada célula de código
`morph.hpp`	Abstração didática de operações de PDI
`stb_image.h` / `stb_image_write.h`	Leitura/escrita de PNG/JPEG (sem OpenCV)
`testsuite.py`	Execução e validação automática dos EPs

1.8 Sobre o morph.hpp

A morph.hpp é uma versão C++ mínima e didática da morph.py: implementa as funções usadas neste capítulo (read, gray, randomImage, show, write, threshold, drawImg), além de operações de morfologia (dil/ero e variantes dil0/ero0/dil1/ero1) preparadas para os capítulos seguintes. Não há paridade numérica garantida com a implementação Python — o critério é "compila e produz uma imagem plausível", não "resultado bit-a-bit idêntico ao Python".

As bibliotecas stb_image.h/stb_image_write.h (domínio público/MIT) são **vendorizadas** junto ao repositório — ou seja, o código-fonte delas já vem copiado para dentro do próprio projeto, em vez de instalado separadamente via apt install — o que evita depender de pacotes do sistema durante a compilação no Colab. As operações dil()/ero() usam cv::dilate/cv::erode quando compiladas com -DMM_USE_OPENCV; sem essa *flag* (padrão, inclusive no Moodle/VPL), caem para a versão didática equivalente (dil1/ero1) sem exigir OpenCV.

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # artefatos de build da trilha C++ (.cpp, binário,
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")

# O kernel é Python mesmo na trilha C++: `mm` (morph.py) é usado pelos
# simuladores, pela exibição das figuras que o binário C++ gera e pelo
# estado mm::Image entre células. cpp=True baixa também a trilha compilada
# (morph.hpp + stb_image*.h), usada no #include das células %%writefile *.cpp.
import config
config.setup(testsuite=True, cpp=True)
from morph import mm
from testsuite import TestSuite
```

```
Ambiente pronto. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

1.9 Fundamentos de Matrizes — atenção à cópia de referências

Como uma imagem digital pode ser representada por uma matriz, é importante compreender como criar e manipular matrizes corretamente. Em C++, std::vector tem **semântica de valor**: copiar o vetor copia os seus dados. Por isso std::vector<std::vector<int>> m(3, std::vector<int>(2, 0)) cria de fato três linhas **independentes** — o construtor de preenchimento copia a linha-modelo três vezes.

> ⚠ Atenção à cópia de referências
>
> A armadilha aparece quando se usa **ponteiros** ou **referências** em vez de cópias. Em std::vector<int> linha(2, 0); std::vector<std::vector<int>*> m(3, &linha);, os três elementos de m apontam para a **mesma** linha, e alterar (*m[0])[0] altera todas. O mesmo ocorre com auto& linha = m[0]; (referência — modifica a matriz), em contraste com auto linha = m[0]; (cópia — deixa a matriz intacta).

Para visualizar esse comportamento, pode-se executar o código no Python Tutor (que também executa C++ passo a passo) e comparar o efeito de cópias e de referências.

Na prática, para o processamento digital de imagens (PDI), utiliza-se a estrutura mm::Image da morph.hpp, que também tem semântica de valor: mm::Image b = a; copia os pixels, enquanto mm::Image& b = a; cria apenas um apelido para a mesma imagem. O código a seguir apresenta diferentes formas de criação de imagens sintéticas, cujos resultados são exibidos na Figura 1.6.

```
%%writefile tmp/fig_imagens_sinteticas.cpp
#define MM_OUT "tmp/fig_imagens_sinteticas.png"
#include "morph.hpp"
#include <algorithm>
#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) << std::endl;

    std::vector<mm::Image> images = {img_preta, img_branca, img_random};
    std::vector<std::string> 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)"
    };

    mm::show(images, MM_OUT, titles, 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:
137 145 160  15 105   7
 74 121 203 251 149 163
 25 188   0 205  92  88
 61 107 130  72 204 150

[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)")
```

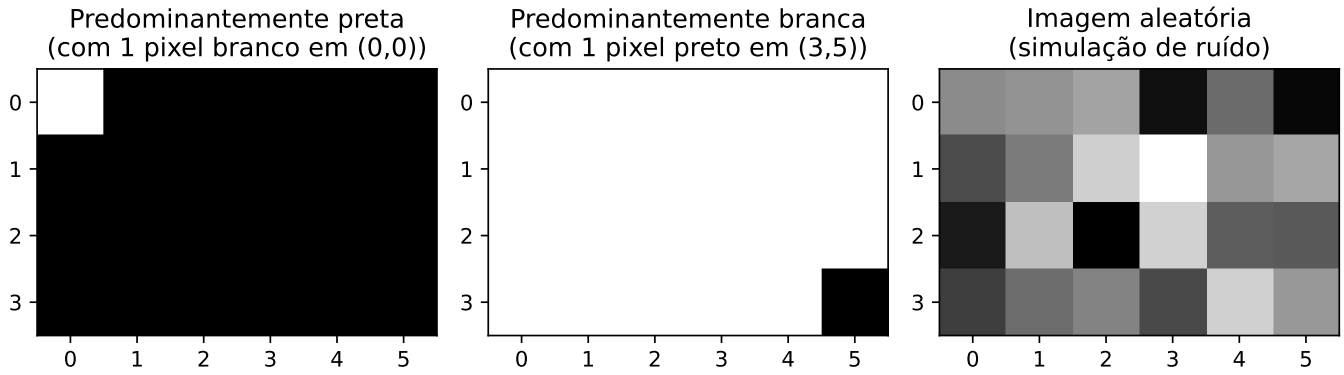


Figura 1.6: Exemplos de imagens sintéticas representadas matricialmente.

1.10 Lendo e Exibindo Imagens

Em bibliotecas como o OpenCV (`cv::Mat`), imagens digitais são representadas computacionalmente como matrizes multidimensionais. Na `morph.hpp`, a estrutura `mm::Image` adota uma abordagem mais simples: um buffer linear de bytes (`std::vector<unsigned char>`), com altura, largura e número de canais explícitos.

Uma das operações fundamentais em PDI é a leitura de imagens.

Na `morph.hpp`, a função `mm::read()` permite carregar imagens tanto de arquivos locais quanto de URLs, como ilustrado na Figura 2.7.

```
%%writefile tmp/fig_01_natureza.cpp
#define MM_OUT "tmp/fig_01_natureza.png"
// Compile: g++ -std=c++17 -o programa programa.cpp -I. -lcurl
#include "morph.hpp"
#include <iostream>
#include <string>
#include <filesystem>

//| label: fig-01-natureza
```

```
//| fig-cap: "Mandrill (Mandrillus sphinx) em ambiente natural. Crédito: Julien Renoult (CC BY
4.0)."
//| echo: true

int main() {
    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;

    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);
    }

    // A imagem já está no formato mm::Image
    mm::Image img = img_obj;

    std::cout << "Dimensões (H, W, Canais): (" << img.h << ", " << img.w << ", " <<
img.channels << ")" << std::endl;
    std::cout << "Tipo de dado: uint8_t" << 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"
```

```
Dimensões (H, W, Canais): (1365, 2048, 3)
Tipo de dado: uint8_t
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)")
```

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

Alternativa: *Download* da Imagem para Armazenamento Local

Em ambientes nos quais a leitura direta de URLs esteja indisponível — devido a restrições de rede, políticas de *firewall* ou ausência de conectividade — uma alternativa consiste em baixar previamente a imagem para o sistema de arquivos local e então carregá-la com `mm::read()`. Na trilha C++, `mm::read` também aceita URLs diretamente (download interno via `curl`/`wget`); esta alternativa cobre o caso de baixar uma vez e reutilizar o arquivo local. No exemplo a seguir, o arquivo é salvo como `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");   // lê do arquivo local
    mm::show(img, MM_OUT);                      // Exemplo: Captura RGB (arquivo local)
    return 0;
}
```

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

Explicação

- `wget -O mandrill.png <URL>` realiza o *download* da imagem e a armazena localmente com o nome especificado;
- `mm::read("mandrill.png")` efetua a leitura do arquivo diretamente do sistema de arquivos, sem necessidade de requisições HTTP adicionais;
- essa estratégia reduz a dependência de conectividade durante a execução dos experimentos e evita *downloads* repetidos da mesma imagem.

Nota

O prefixo `!` é utilizado em ambientes baseados em *notebooks*, como Jupyter Notebook, JupyterLab e Google Colab, para executar comandos do sistema operacional diretamente em células de código. Em terminais convencionais, o comando deve ser utilizado sem o prefixo `!`.
Caso o utilitário `wget` não esteja instalado, pode-se utilizar alternativamente:

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

1.11 Conversão de Tipos e Limiarização

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

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

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

Conversão para Tons de Cinza

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

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

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

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

Limiarização (*Thresholding*)

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

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

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

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

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

Exemplo prático de conversão e limiarização

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

```
%%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]

    // img já está inicializado (fornecido automaticamente)

    // 1. Converter para Tons de Cinza
    mm::Image img_gray = mm::gray(img);

    // 2. Aplicar limiar (Pixels > 128 tornam-se 255, outros 0)
    int limiar = 128;
    mm::Image img_binaria = mm::threshold(img_gray, limiar);

    // Uso da nova função
    mm::show(
        {img, img_gray, img_binaria},
        MM_OUT,
        {"Original", "Tons de Cinza", "Binária (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] Tons de Cinza
[3] Binária (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)")
```

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

1.12 Limiarização pelo método de Otsu

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

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

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

O código abaixo compara a limiarização manual ($T = 128$) com a automática (Otsu). A binarização por Otsu é feita com `mm::threshold(img_gray)`; como a API mínima da `morph.hpp` não devolve o T calculado, o valor exibido no rótulo da figura é recuperado com `cv2.threshold` na etapa de exibição (mesmo algoritmo, mesmo T).

```
%%writefile tmp/fig_01_otsu.cpp
#define MM_OUT "tmp/fig_01_otsu.png"
#include <iostream>
#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_38.png");
// [pdi:state-io:end]

    // img_gray já é fornecida (mm::Image)

    // 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)
    int T_otsu = mm::otsu(img_gray);  // calcula T via Otsu
    mm::Image img_bin_otsu = mm::threshold(img_gray);  // usa o mesmo T de Otsu
    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=128)", "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)
```

```
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)")
```

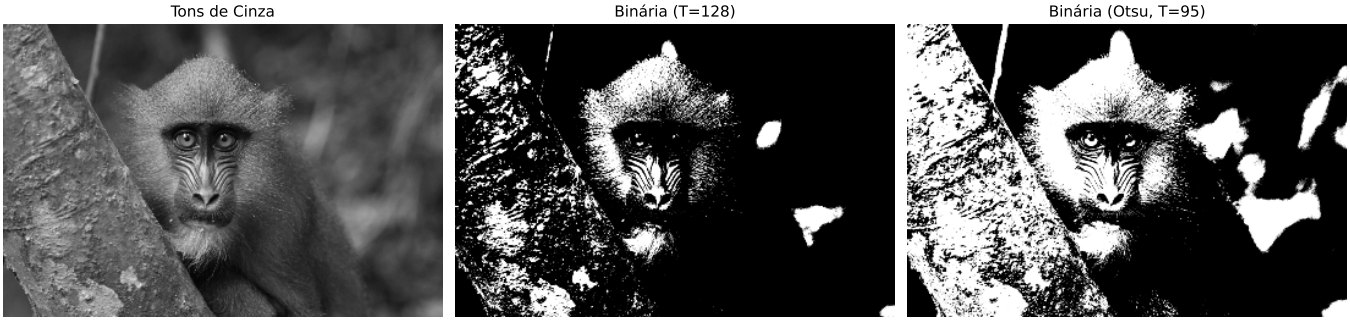


Figura 1.9: Comparação entre limiarização manual (T=128) e automática (Otsu) sobre a imagem em tons de cinza.

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

Simplicidade da biblioteca `morph.hpp`

Enquanto o OpenCV exige a chamada completa:

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

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

1.13 Acesso a Pixels

Na `morph.hpp`, a imagem é um `mm::Image` com um buffer linear `data` (linha após linha, canais intercalados) e o método `at(y, x, c)` para acesso individual, seguindo a convenção matricial **linha** (eixo Y) e **coluna** (eixo X): `img.at(linha, coluna)` para tons de cinza e `img.at(linha, coluna, canal)` para cada canal de uma imagem RGB.

O código da Figura 1.10 demonstra como extrair esses valores em imagens coloridas (RGB) e em tons de cinza, além de isolar a vizinhança imediata do ponto de interesse.

Acesso a pixel em C++ (`mm::Image`):

- **Tons de cinza:** `img_gray.at(r, c)` devolve um `unsigned char` (0 a 255) com a intensidade do cinza no pixel.
- **RGB:** `img.at(r, c, 0)`, `img.at(r, c, 1)`, `img.at(r, c, 2)` acessam os canais **R**, **G** e **B** — um índice de canal por vez; não há vetor `[R, G, B]`.
- **Vizinhança** 3×3**:** percorrida por dois laços aninhados sobre `img_gray.at(r + dy, c + dx)`, com `dy, dx` em $\{-1, 0, 1\}$ — não há fatiamento (*slicing*) como em NumPy. Essa varredura é a base para filtros espaciais e convoluções.

```
# Ainda não portado para esta linguagem nesta versão - referência conceitual em Python.

# 1. Coordenadas do pixel alvo (linha, coluna)
r, c = 600, 800

# 2. Acesso direto aos valores do pixel
pixel_cinza = img_gray[r, c]
print(f"Pixel alvo ({r}, {c}):")
print(f"  Tons de cinza (escalar) : {pixel_cinza}")
print(f"  Colorido (canais RGB)   : R={img[r, c, 0]}, G={img[r, c, 1]}, B={img[r, c, 2]}")


# 3. Vizinhanca 3x3 ao redor de (r, c); o pixel central vai entre colchetes
print("Matriz de vizinhanca 3x3 (tons de cinza):")
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. Marca a posicao do pixel com um quadrado vermelho vazado (borda de
#    3 px) numa copia da imagem e exibe (equivalente ao ponto do matplotlib).
marcada = img.copy()
lado = 20   # meio-lado do quadrado, em pixels
esp = 5     # espessura da borda

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"Localizacao do pixel ({r}, {c})")
```

Figura 1.10

- Não há plotagem interativa (equivalente a `plt.plot`): para marcar a posição do pixel, a célula de código a seguir **desenha um quadrado vermelho vazado** ao redor dele numa cópia da imagem (`marcada = img.copy()`, depois `marcada.at(...)`) e exibe com `mm::show`.

A indexação é *zero-based*: `(0,0)` é o canto superior esquerdo. A primeira dimensão controla a **altura** (linhas/Y) e a segunda a **largura** (colunas/X).

1.14 Resumo

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

- **Imagem digital** = função $f(x, y)$ que mapeia coordenadas para intensidades (escalares ou vetoriais).
- **Domínio**: conjunto finito $\mathbb{E} = \{(x, y) \in \mathbb{Z}^2 \mid 0 \leq x < L,\ 0 \leq y < H\}$.
- **Tipos principais**: binária ($\mathcal{V} = \{0, 255\}$), tons de cinza ($\mathcal{V} = [0, 255]$) e RGB ($\mathcal{V} = [0, 255]^3$).
- **Limiarização** converte tons de cinza em binária; o **método de Otsu** determina o valor de corte automaticamente maximizando a variância entre classes.
- A `morph.hpp` (ou `mm::`) oferece funções didáticas para operações básicas de PDI, como `mm::gray()`, `mm::threshold()`, `mm::show()` (sobrecarregada para 1 imagem ou várias).
- **Armadilha de cópia**: `std::vector` tem semântica de valor, mas ponteiros/referências compartilhados entre "linhas" de uma matriz reintroduzem o mesmo problema do NumPy — prefira `mm::Image`, que também copia por valor.
- Acesso a pixels via `img.at(linha, coluna)`, com indexação ***zero-based***.

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

1.15 🤖 Uso do Gemini Notebook como Tutor Complementar

Nesta edição, além dos *notebooks* interativos no Google Colab, o **Gemini Notebook** está disponível como ferramenta complementar de estudo. A plataforma utiliza exclusivamente os documentos fornecidos pelo autor como base de conhecimento, garantindo respostas coerentes com o conteúdo do livro.

🎓 Estude com o Tutor Inteligente

Acesse o ambiente do capítulo pelo *link* abaixo e explore especialmente as opções **Guia de Estudo** e **Conversa** para aprofundar sua compreensão.
🚀 ACESSAR Gemini Notebook: CAPÍTULO 01

🌐 Idioma e Linguagem de Programação

O projeto deste capítulo no Gemini Notebook foi construído apenas com o texto em **português** e os exemplos de código em **Python**. Se você está estudando pela edição em inglês ou francês, ou acompanhando a trilha em C++, as respostas do tutor podem não corresponder exatamente à versão que você está lendo.

⚠️ Aviso sobre Conteúdo Gerado por IA

A IA é uma poderosa aliada nos estudos, mas o conteúdo gerado pode conter **erros ou imprecisões**. Consulte sempre **livros, artigos científicos e outras fontes acadêmicas confiáveis** para validar as informações. Sempre que possível, execute os exemplos práticos fornecidos neste capítulo para verificar os resultados.

Funcionalidades Disponíveis na Plataforma

O **Gemini Notebook** oferece uma suíte avançada de ferramentas baseadas em IA para transformar o conteúdo estático do livro em uma experiência de aprendizado dinâmica e multimídia. A plataforma utiliza técnicas de **RAG** (*Retrieval-Augmented Generation*), fundamentadas no trabalho de Lewis (2020), para basear as respostas estritamente nos documentos fornecidos e minimizar a ocorrência de alucinações.

As principais funcionalidades incluem:

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

1.16 Lista de Exercícios

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

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

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

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

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

4. (20%) O que é **limiarização** (*thresholding*)? Explique a diferença entre escolher um limiar T fixo (ex.: $T = 128$) e utilizar o **método de Otsu** para determinação automática do limiar. Em poucas palavras, como o método de Otsu escolhe o limiar?

5. (15%) No contexto da biblioteca didática `mm` discutida no capítulo, responda:

   - a) (7,5%) Como se acessa o valor do pixel na posição (linha=50, coluna=60) de uma imagem em tons de cinza `img_gray`?
   - b) (7,5%) Qual é a vantagem de usar `mm::threshold(img_gray)` sem passar o limiar? Compare com a chamada equivalente no OpenCV (`cv::threshold`).

6. (15%) O que os campos `h`, `w` e `channels` de um `mm::Image` representam para uma imagem RGB? Dê um exemplo concreto com uma imagem de 640×480 pixels.

Referências do Capítulo

A fundamentação teórica deste capítulo compreende as seguintes obras de PDI e VC:

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

1.17 💻 Parte Prática com Exercícios de Programação

🎯 Objetivo deste Caderno

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

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

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

§ Instruções Passo a Passo

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

Preparação do Ambiente

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

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

Escrevendo o Código

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

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

Download

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

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # artefatos de build da trilha C++ (.cpp, binário,
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")

# O kernel é Python mesmo na trilha C++: `mm` (morph.py) é usado pelos
# simuladores, pela exibição das figuras que o binário C++ gera e pelo
# estado mm::Image entre células. cpp=True baixa também a trilha compilada
# (morph.hpp + stb_image*.h), usada no #include das células %%writefile *.cpp.
import config
config.setup(testsuite=True, cpp=True)
from morph import mm
from testsuite import TestSuite
```

```
Ambiente pronto. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Executando os Testes

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

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

Substitua a extensão conforme a linguagem usada:

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

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

Para testar código Python diretamente, sem salvar arquivo, use `run_code(codigo)` passando o código como *string* numa variável `codigo`:

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

⚠️ Importante: Regras e Boas Práticas

🔹 Sobre a Entrada de Dados

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

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

🔹 Configuração de IA no Colab

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

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

🔹 **Integridade Acadêmica (Plágio)**

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

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

1.17.1 EP01_01 📏 Três métricas de distância em PDI

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

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

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

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

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

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

📌 **Importante**:

- Utilize as funções matemáticas padrão da sua linguagem: `math.sqrt`, `abs` (ou `fabs`) e `max`.
- A saída deve conter **apenas os números** (um por linha), sem textos adicionais.
- Ver um simulador interativo para esta questão na **Figura 1.11** (gráfico com arrasto dos pontos e visualização das três métricas).

1.17.1.1 🖼️ Por que isso importa? – Custo computacional

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

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

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

1.17.1.2 📋 Tarefa (especificação para VPL)

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

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

1.17.1.3 📌 Exemplos

Entrada	Saída	Observação
0 0 3 4	5.00 7.00 4.00	Triângulo 3-4-5
0 0 1 1	1.41 2.00 1.00	Diagonal unitária

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

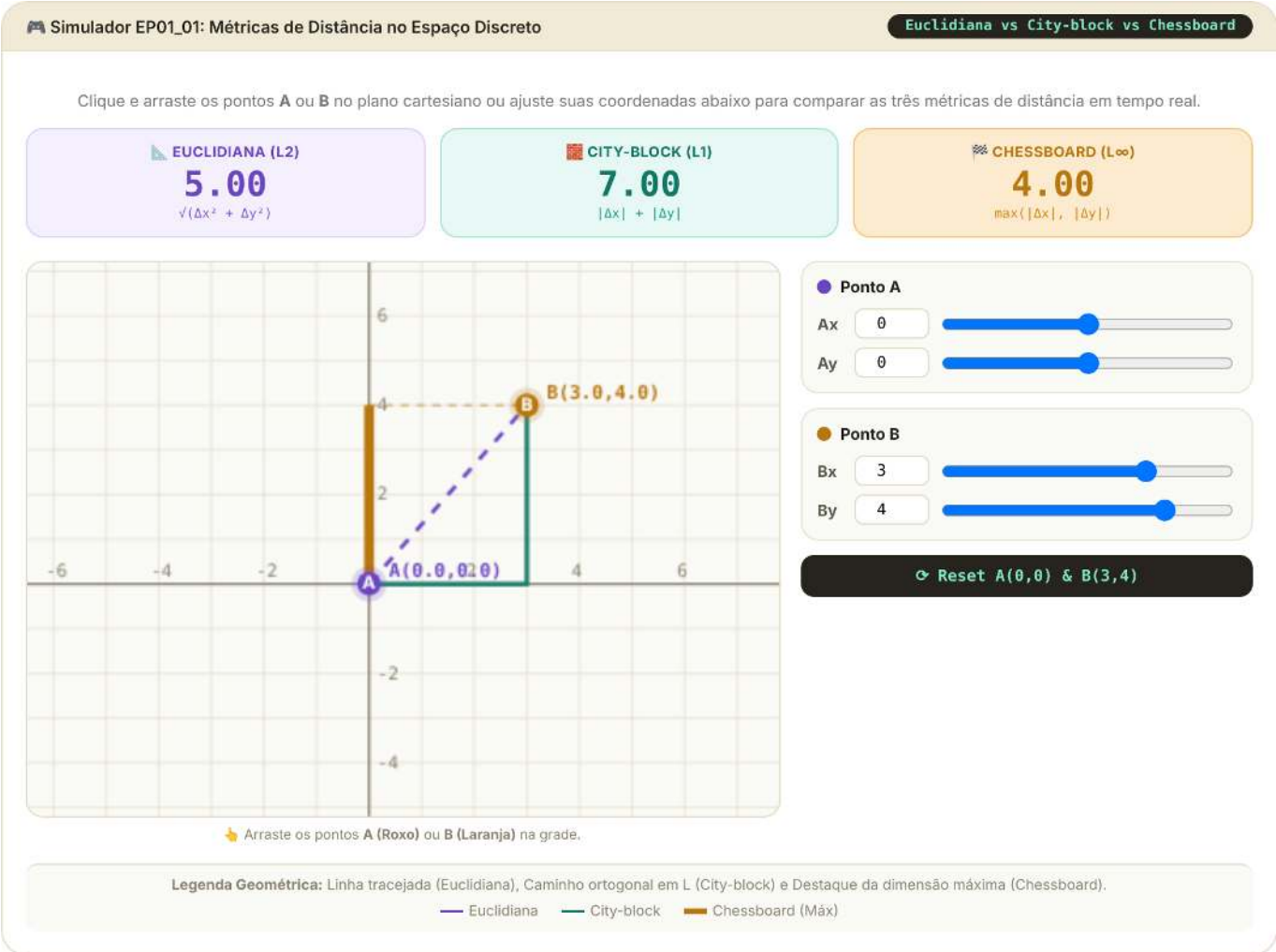
```
%%writefile tmp/mm_out_1.cpp
#include <iostream>
#include <cmath>
#include <chrono>

// Compile: g++ -O2 -o benchmark benchmark.cpp -lm

constexpr 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() {
    // Medição da soma simples
    auto start_soma = std::chrono::high_resolution_clock::now();
    volatile double resultado_soma = 0;
    for (int i = 0; i < N; ++i) {
        resultado_soma = apenas_soma();
    }
    auto end_soma = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> t_soma = end_soma - start_soma;

    // Medição da soma + sqrt
    auto start_sqrt = std::chrono::high_resolution_clock::now();
    volatile double resultado_sqrt = 0;
    for (int i = 0; i < N; ++i) {
        resultado_sqrt = soma_e_sqrt();
    }
    auto end_sqrt = std::chrono::high_resolution_clock::now();
```

```
    std::chrono::duration<double> t_sqrt = end_sqrt - start_sqrt;

    std::cout << "Soma simples       : " << t_soma.count() << " s" << std::endl;
    std::cout << "Soma + sqrt        : " << t_sqrt.count() << " s" << std::endl;
    std::cout << "Razão (sqrt/soma)  : " << t_sqrt.count() / t_soma.count() << "x" <<
std::endl;

    // Evita otimização do compilador
    std::cout << "Resultados: " << resultado_soma << " " << resultado_sqrt << std::endl;

    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       : 0.156818 s
Soma + sqrt        : 0.216574 s
Razão (sqrt/soma)  : 1.38105x
Resultados: 7 5
```



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

Figura 1.11: Simulador EP01_01: Distâncias Euclidiana, City-block e Chessboard

1.17.1.4 🐍 Python

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

Exemplo de célula:

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

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

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

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

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

```
Overwriting EP01_01.py
```

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

# %run -i EP01_01.py
```

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

```
5.66
8.00
4.00
```

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

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

1.17.1.5 ☕ Java

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

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

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

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

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

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

```
Overwriting EP01_01.java
```

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

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

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

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

1.17.1.6 💻 C

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

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

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

instalar_gcc()
```

```
  GCC já disponível.
```

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

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

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

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

    return 0;
}
```

```
Overwriting EP01_01.c
```

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

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

```
5.66
8.00
4.00
```

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

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

1.17.1.7 💻 C++

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

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

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

instalar_gpp()
```

```
  G++ já disponível.
```

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

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

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

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

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

    return 0;
}
```

```
Overwriting EP01_01.cpp
```

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

```
5.66
8.00
4.00
```

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

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

1.17.1.8 🌐 JavaScript (Node.js)

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

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

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

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

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

```
Overwriting EP01_01.js
```

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

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

1.17.1.9 📊 R

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

Exemplo de célula:

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

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

def instalar_r():
    if shutil.which("R"):
        print("  R já disponível."); 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("  Ambiente R pronto.")

instalar_r()
```

```
  R já disponível.
```

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

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

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

```
Overwriting EP01_01.r
```

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

```
5.66
8.00
4.00
```

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

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

1.17.2 EP01_02 📊 Desempenho Preditivo — Métricas de ML em VC

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

1.17.2.1 🧠 Por que a métrica certa importa?

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

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

1.17.2.2 📊 A Matriz de Confusão

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

Tarefa:

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

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

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

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

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

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

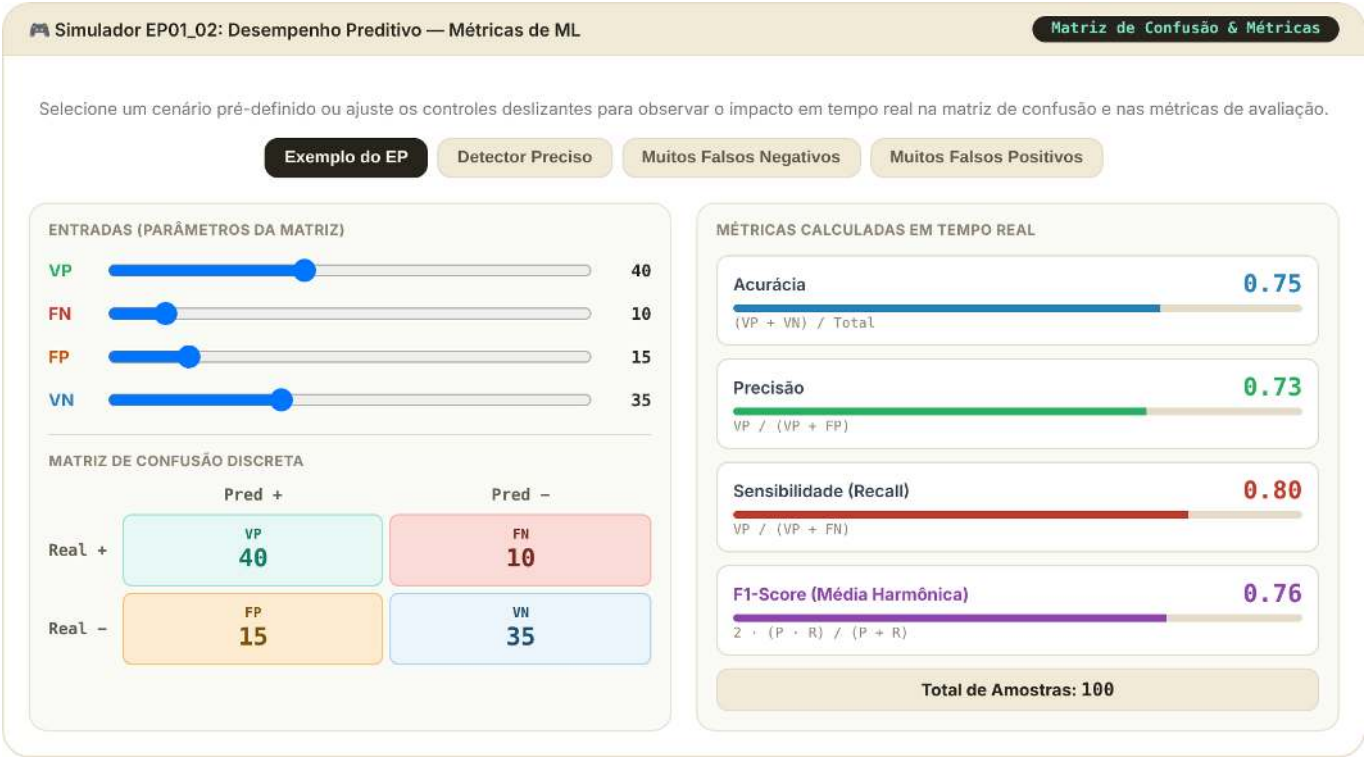
📌 **Importante**:

- Use divisão em ponto flutuante para evitar resultados truncados.
- A ordem de saída deve ser: **Acurácia**, **Precisão**, **Sensibilidade** e **F1-score**.
- Veja a simulação interativa deste EP na Figura 1.12.

1.17.2.3 📌 Exemplo de Execução

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

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



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

Figura 1.12: Simulador EP01_02: Desempenho Preditivo — Métricas de ML

```
%%writefile EP01_02.cpp
// sua solução
```

```
Overwriting EP01_02.cpp
```

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

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

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

Nesta atividade, você avaliará um classificador binário (ex.: detecção de desmatamento em imagens de satélite, ver dgi.inpe.br) através da **curva Precisão-Sensibilidade** e da métrica **mAP (*Mean Average***

***Precision*)**. O mAP é padrão em competições como COCO (*Common Objects in Context*) e PASCAL VOC (*Visual Object Classes*) e dos modelos YOLO (*You Only Look Once*).

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

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

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

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

Referências: Roboflow — mAP · Vídeo explicativo

1.17.3.2 Como o mAP é calculado — passo a passo

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

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

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

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

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

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

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

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

1.17.3.3 📋 Tarefa

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

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

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

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

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

1.17.3.4 📌 Importante

- Limiar fixo para as métricas individuais: **0.85**
- Divisão segura: se denominador for zero, use 0
- Formatação: duas casas decimais
- Monotonize de trás para frente
- A Figura 1.13 apresenta uma simulação desta questão

1.17.3.5 📌 Exemplo de Execução

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

1.17.3.6 🐍 Dica para calcular o AP (com regra do trapézio)

```
def calcular_AP(verdades, confiancas, limiares):
    m = len(limiares)
    precisoes = [0.0] * m
    sensibilidades = [0.0] * m
    for i in range(m):
        p, s = calcular_metricas(verdades, confiancas, limiares[i])
        precisoes[m-1-i] = p
        sensibilidades[m-1-i] = s
    prec_mono = precisoes.copy()
    for i in range(m-2, -1, -1):
        if prec_mono[i] < prec_mono[i+1]:
            prec_mono[i] = prec_mono[i+1]
    AP = 0.0
    for i in range(1, m):
```

```
        # Regra do trapézio: média das alturas vezes a base
        area_trapezio = (prec_mono[i-1] + prec_mono[i]) / 2.0
        AP += area_trapezio * (sensibilidades[i] - sensibilidades[i-1])
    return precisoes, prec_mono, sensibilidades, AP
```

```
%%writefile EP01_03.cpp
// sua solução
```

```
Overwriting EP01_03.cpp
```

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

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

1.17.4 EP01_04 🖼️ Leitura e Informações de uma Imagem em Matriz

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

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

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

📌 **Importante**:

- A saída deve seguir exatamente o formato rotulado (ex: `Linhas: X`).
- O valor da média deve ser formatado com **duas casas decimais**.
- Ver um simulador interativo para esta questão na **Figura 1.14** (grade interativa para visualização de intensidades e cálculos em tempo real).

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

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

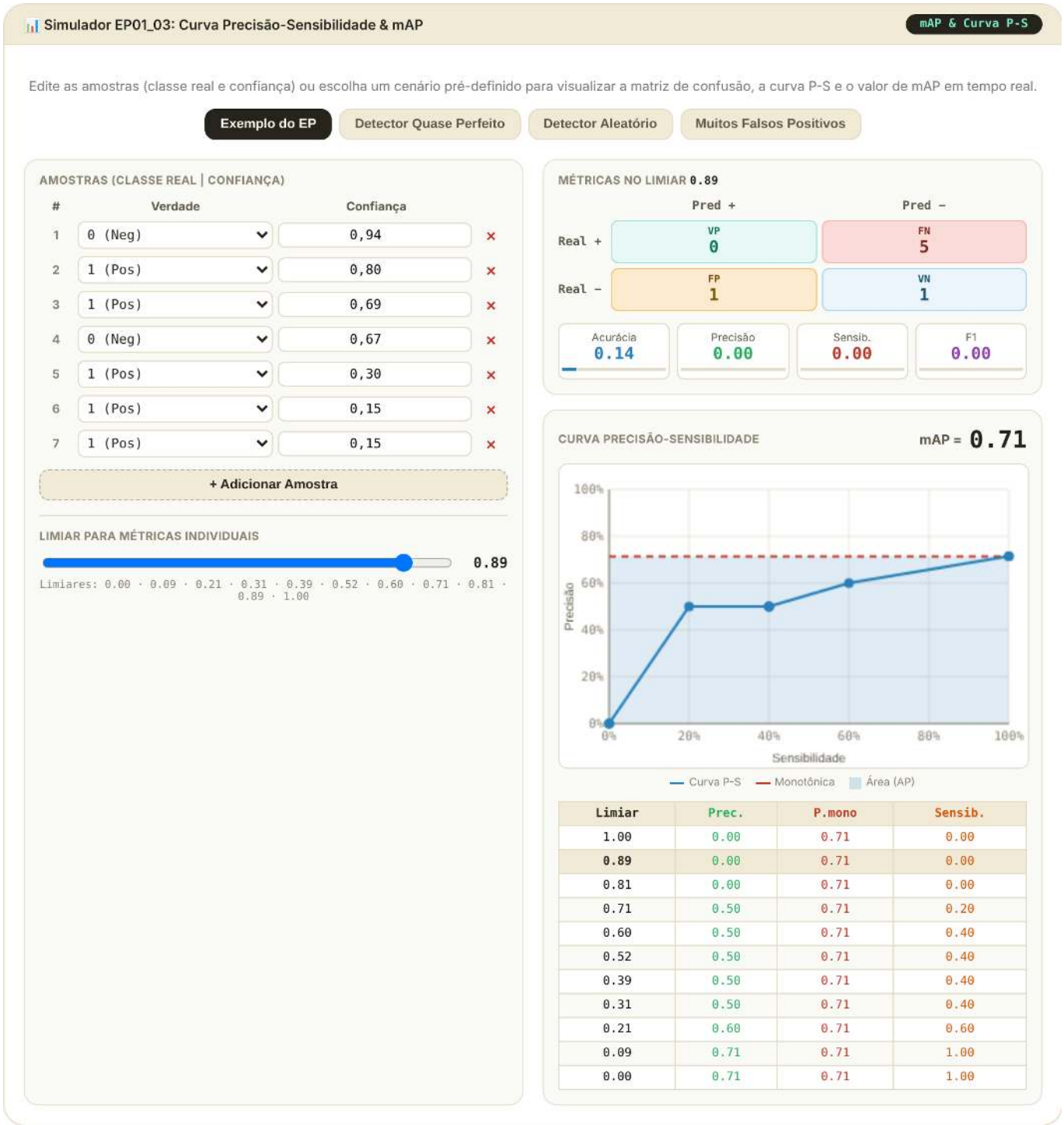
Operação	Utilidade Prática
Máximo/Mínimo	Identificar se a imagem está “lavada” (baixo contraste) ou saturada.
Média	Calcular o brilho global da cena para ajustes de exposição.
Normalização	Redimensionar os valores para intervalos como $[0, 1]$ em redes neurais.

1.17.4.2 📋 Tarefa (especificação para VPL)

Entrada:

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

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



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

Figura 1.13: Simulador EP01_03: Mean Average Precision (mAP) e Curva P-S

As linhas seguintes contêm os elementos da matriz.

Saída:

Cinco linhas formatadas conforme o exemplo:

`Linhas: L`

`Colunas: C`

`Max: V`

`Min: V`

`Media: V.VV`

1.17.4.3 📌 Exemplos

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

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

Figura 1.14: Simulador EP01_04: Estatísticas de Pixels em Matriz Discreta 5x5

```
%%writefile EP01_04.cpp
// sua solução
```

```
Overwriting EP01_04.cpp
```

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

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

1.17.5 EP01_05 🔄 Negativo de uma Imagem em Tons de Cinza

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

- Leia dois inteiros **L** e **C**, representando o número de linhas e colunas.
- Leia os valores inteiros que compõem a matriz da imagem.
- Para cada pixel, aplique a transformação de inversão:

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

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

📌 **Importante**:

- Os valores de cada linha na saída devem ser separados por um **espaço em branco**.
- A saída deve conter **apenas os números** da matriz resultante.
- Ver um simulador interativo para esta questão na **Figura 1.15** (comparação em tempo real entre a matriz original e seu negativo).

1.17.5.1 🧠 Por que isso importa? – Inversão de Intensidade

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

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

1.17.5.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém o inteiro `L`.

A segunda linha contém o inteiro `C`.

As linhas seguintes contêm os elementos da matriz.

Saída:

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

1.17.5.3 📌 Exemplos

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

```
%%writefile EP01_05.cpp
// sua solução
```

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

Figura 1.15: Simulador EP01_05: Transformação de Negativo de Imagem (Inversão de Intensidade)

```
Overwriting EP01_05.cpp
```

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

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

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

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

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

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

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

📌 **Importante**:

- Utilize a função `round()` da sua linguagem para garantir o arredondamento correto para o inteiro mais próximo.
- A saída deve conter apenas os valores de cinza, mantendo a estrutura de matriz (separados por espaço na linha).
- Ver um simulador interativo para esta questão na **Figura 1.16** (ajuste os sliders para ver como cada cor contribui para o brilho final).

1.17.6.1 🧠 Por que não usar apenas a média?

O olho humano não percebe todas as cores com a mesma intensidade. Somos muito mais sensíveis ao **Verde** do que ao **Azul** devido à nossa evolução biológica. O padrão ITU-R BT.601 utiliza pesos específicos para

criar uma imagem em cinza que pareça naturalmente correta para nossa visão:

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

1.17.6.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém o inteiro `L`.

A segunda linha contém o inteiro `C`.

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

Saída:

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

1.17.6.3 📌 Exemplos

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

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

Figura 1.16: Simulador EP01_06: Conversão RGB para Grayscale (Ponderação Perceptual ITU-R BT.601)

```
%%writefile EP01_06.cpp
// sua solução
```

```
Overwriting EP01_06.cpp
```

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

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

1.17.7 EP01_07 ⚫ Limiarização Manual: Imagem Binária

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

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

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

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

📌 **Importante**:

- Preste atenção ao operador: o pixel só vira branco (255) se for **estritamente maior** que T.
- A saída deve manter a estrutura de matriz (L linhas e C colunas).
- Ver um simulador interativo para esta questão na **Figura 1.17** (ajuste o slider de T para observar como os objetos são isolados do fundo).

1.17.7.1 🧠 O que é Segmentação?

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

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

1.17.7.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém o inteiro `L`.

A segunda linha contém o inteiro `C`.

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

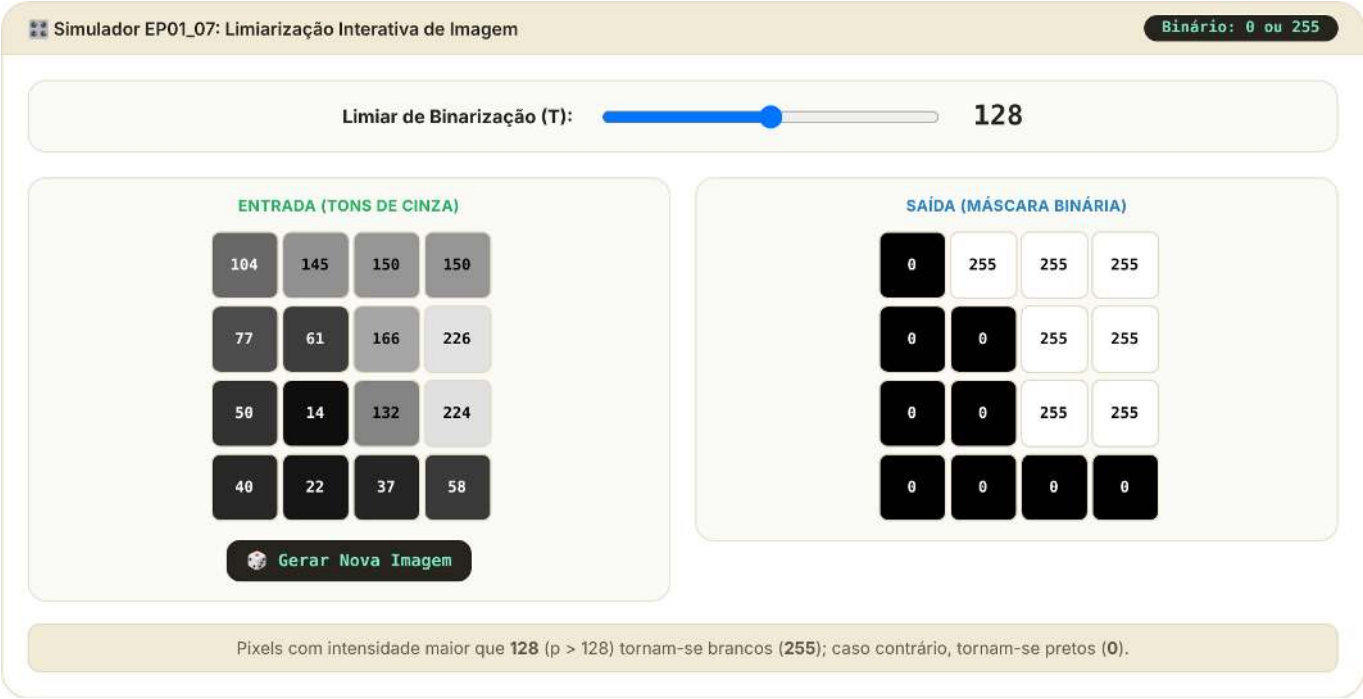
As linhas seguintes contêm os elementos da matriz.

Saída:

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

1.17.7.3 📌 Exemplos

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



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

Figura 1.17: Simulador EP01_07: Limiarização Global Interativa (Binarização p > T)

```
%%writefile EP01_07.cpp
// sua solução
```

```
Overwriting EP01_07.cpp
```

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

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

1.17.8 EP01_08 🎨 Remapeamento por Faixa de Intensidade

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

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

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

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

📌 **Importante**:

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

1.17.8.1 🧠 Transformação Condicional de Pixels

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

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

1.17.8.2 📋 Tarefa (especificação para VPL)

Entrada:

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

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

As linhas seguintes contêm os elementos da matriz.

Saída:

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

1.17.8.3 📌 Exemplos

Entrada	Saída	Observação

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

```
%%writefile EP01_08.cpp
// sua solução
```

```
Overwriting EP01_08.cpp
```

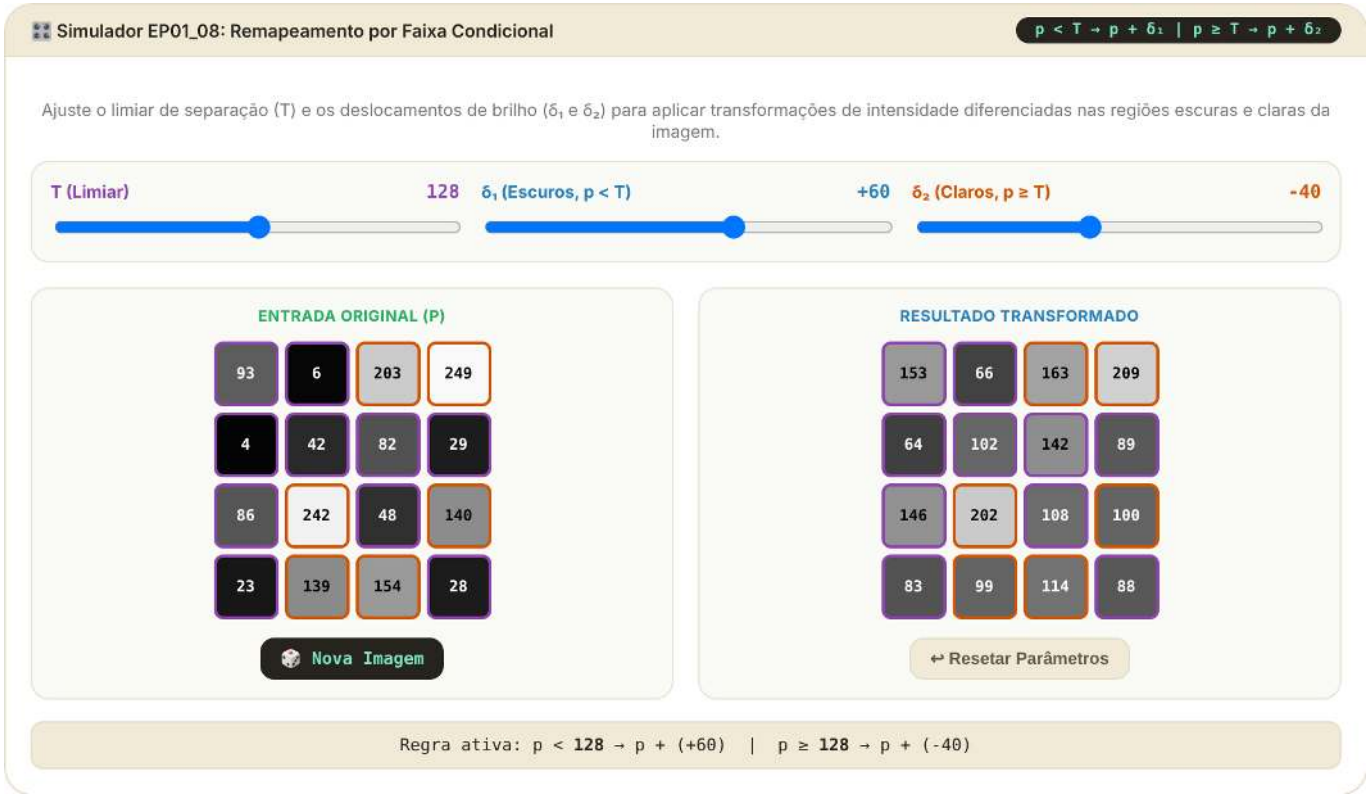
```
TestSuite("EP01_08.cpp").run()
```

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

1.17.9 EP01_09 🏁 Padrão de Tabuleiro: Matriz de Xadrez

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

- Leia dois inteiros **L** (linhas) e **C** (colunas).



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

Figura 1.18: Simulador EP01_08: Remapeamento por Faixa Condicional (Brilho e Contraste por Limiar)

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

📌 **Importante**:

- As cores devem alternar corretamente tanto horizontalmente quanto verticalmente.
- A saída deve ser a matriz impressa linha por linha, com os elementos separados por um espaço.
- Ver um simulador interativo para esta questão na **Figura 1.19** (ajuste as dimensões para visualizar a construção da malha e a saída textual correspondente).

1.17.9.1 🧠 Padrões Sintéticos

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

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

1.17.9.2 📋 Tarefa (especificação para VPL)

Entrada:

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

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

Saída:

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

1.17.9.3 📌 Exemplos

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

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

Figura 1.19: Simulador EP01_09: Gerador de Padrão Xadrez (Lógica de Paridade (i + j) % 2)

```
%%writefile EP01_09.cpp
// sua solução
```

```
Overwriting EP01_09.cpp
```

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

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

1.17.10 EP01_10 📄 Metadados: Leitura de Arquivo PGM

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

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

1. **Versão:** O identificador `P2`.
2. **Comentários:** Linhas opcionais que começam com `#` (devem ser ignoradas).
3. **Dimensões:** Dois inteiros representando **Largura** e **Altura**.
4. **Máximo:** Um inteiro representando a intensidade máxima (geralmente 255).

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

📌 **Importante**:

- **Leitura de Arquivo:** Você deve abrir o arquivo indicado no exemplo utilizando a função `open()` do Python.
- **Ordem de Saída:** Ao contrário da ordem presente no arquivo, a saída esperada deve estar no formato de tupla: `(Altura, Largura, Canais)`.
- Como arquivos PGM são em tons de cinza, o número de **Canais** é sempre **1**.
- Ver um simulador interativo para esta questão na **Figura 1.20** (ajuste as dimensões para ver como o cabeçalho ASCII é gerado).

1.17.10.1 🧠 Entendendo o formato PGM

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

Componente	Exemplo	Significado
Magic Number	`P2`	Identifica que é um PGM em formato de texto (ASCII).
Comentário	`# CREATOR...`	Linha informativa ignorada pelo processador.
Dimensões	`397 343`	**397** colunas (Largura) e **343** linhas (Altura).
Intensidade	`255`	Define o valor do branco puro (escala de 0 a 255).

1.17.10.2 📋 Tarefa (especificação para VPL)

Entrada:

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

Saída:

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

1.17.10.3 📌 Exemplos

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

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

Figura 1.20: Simulador EP01_10: Estrutura de Arquivo PGM (Cabeçalho ASCII P2 e Mapeamento para Tupla Python)

Nota

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

```
%%writefile tmp/mm_out_2.cpp
#include <iostream>
#include <string>
#include <fstream>
#include <cstdlib>
#include <sys/stat.h>

// Função para verificar se um arquivo existe
bool fileExists(const std::string& filename) {
    struct stat buffer;
    return (stat(filename.c_str(), &buffer) == 0);
}

// Função para baixar um arquivo usando curl (substituto de urllib.request)
int downloadFile(const std::string& url, const std::string& filename) {
    std::string command = "curl -s -o " + filename + " -A 'Mozilla/5.0' " + url;
    return system(command.c_str());
}

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::string files[] = {file};

    for (const auto& f : files) {
        if (!fileExists(f)) {
            std::string url = BASE_URL + "/" + f;
            int result = downloadFile(url, f);

            if (result == 0) {
```

```
                std::cout << "  Arquivo baixado: " << f << std::endl;
            } else {
                std::cerr << "  Erro: Não foi possível baixar de " << 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
// sua solução
```

```
Overwriting EP01_10.cpp
```

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

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

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

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

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

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

📌 **Importante**:

- **Bordas:** Nas extremidades do vetor (índices 0 e $n-1$), a vizinhança possui apenas dois elementos (o próprio e o único vizinho disponível). No índice 0, compare apenas $v1[0]$ e $v1[1]$. No último índice, compare apenas $v1[n-2]$ e $v1[n-1]$.
- **Saída:** Imprima o cabeçalho "v2:" seguido pelos valores do vetor resultante, um por linha.
- Ver um simulador interativo para esta questão na **Figura 1.21** (passe o mouse sobre os resultados para visualizar a janela de vizinhança utilizada no cálculo).

1.17.11.1 🧠 Por que analisar vizinhos?

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

Função	Efeito Visual
Realce	Expande estruturas brilhantes e "engorda" objetos claros.
Remoção de Ruído	Elimina pequenos pontos pretos (ruído de "sal e pimenta" escuro).

Função	Efeito Visual
Preenchimento	Fecha pequenos buracos ou lacunas em formas binárias.

1.17.11.2 📋 Tarefa (especificação para VPL)

Entrada:

Um inteiro `n`.

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

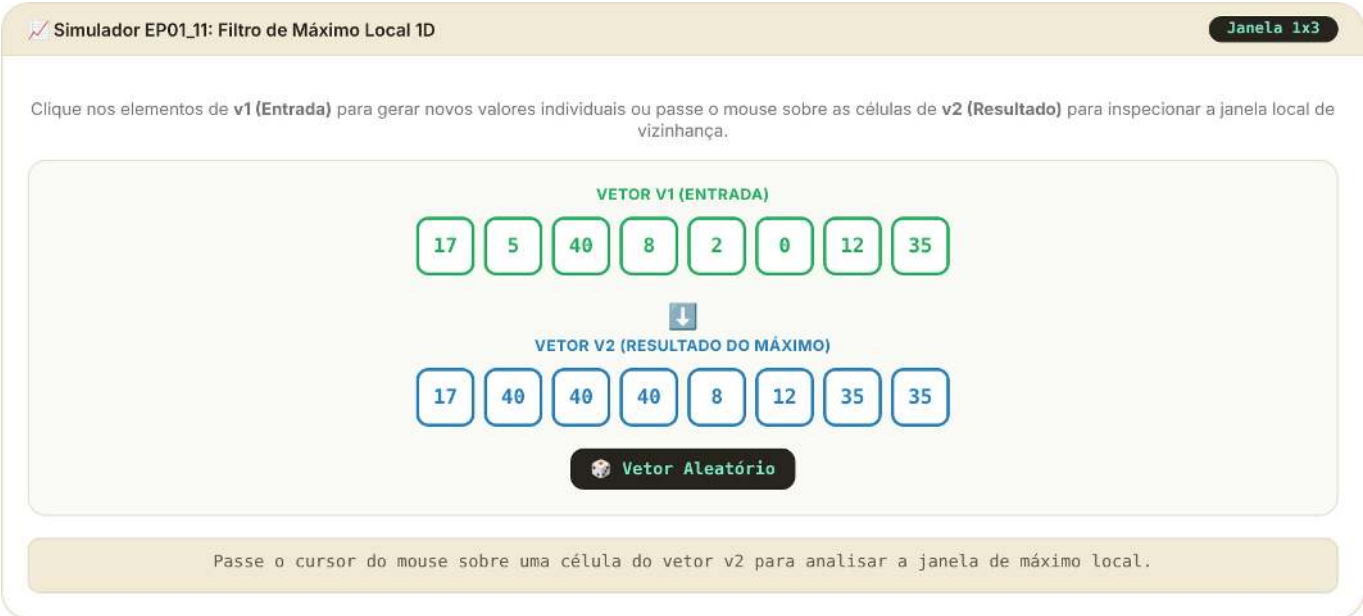
Saída:

A *string* `v2:` na primeira linha.

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

1.17.11.3 📌 Exemplos

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



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

Figura 1.21: Simulador EP01_11: Filtro de Máximo Local 1D (Vizinhança 1x3 com Condição de Borda)

```
%%writefile EP01_11.cpp
// sua solução
```

```
Overwriting EP01_11.cpp
```

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

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

Capítulo 2

Da Captura ao Pixel - Amostragem, Quantização e Conectividade

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

2.1 Objetivos

Ao final deste capítulo, você será capaz de:

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

2.2 O Olho e a Câmera - Elementos da Percepção Visual

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

2.2.1 Visão humana

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

2.2.2 Sensores digitais

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

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

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

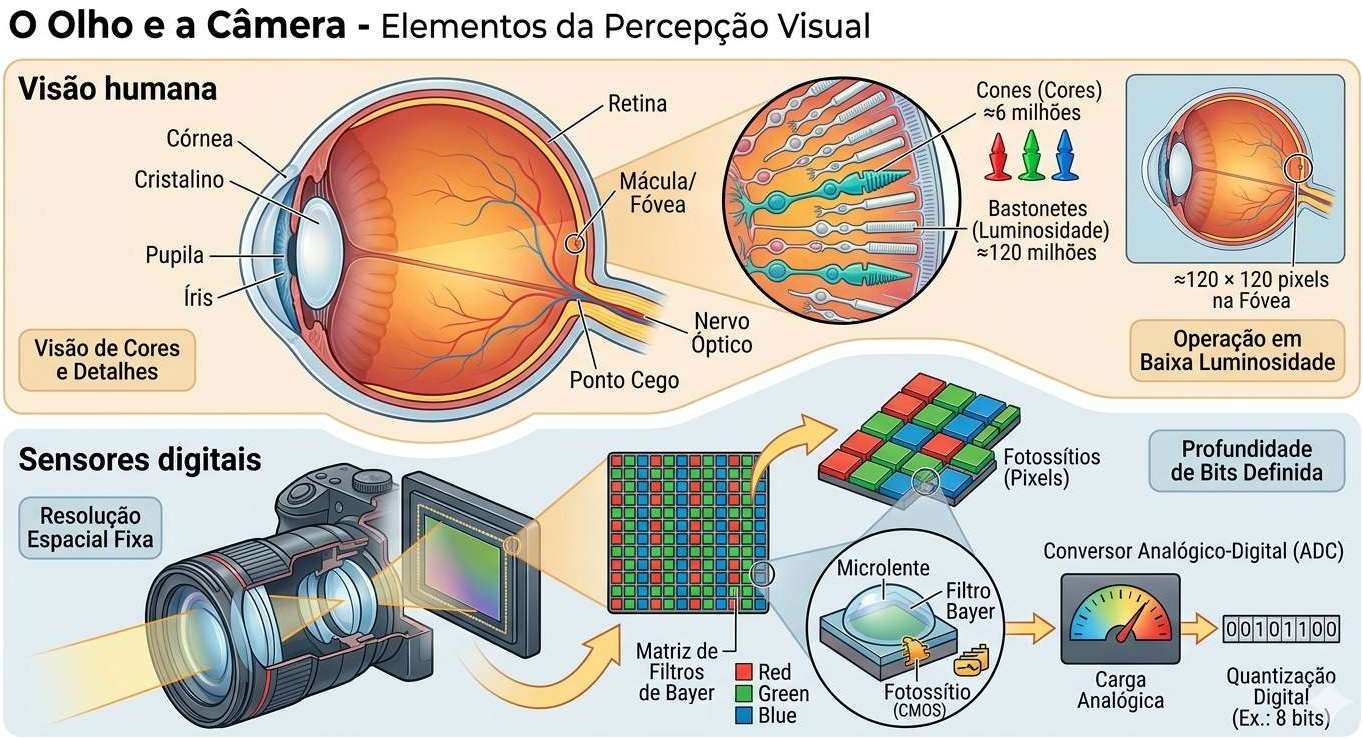


Figura 2.1: Comparativo didático entre o sistema visual biológico e o eletrônico: no topo, a anatomia do olho humano destacando a retina e os fotorreceptores (cones e bastonetes); abaixo, a estrutura de uma câmera digital detalhando o sensor CMOS, a matriz de filtros de Bayer (RGGB) e o processo de quantização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

2.3 Ilusões de Ótica: Os Desafios da Percepção Visual

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

2.3.1 Ambiguidade e Contexto

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

2.3.2 Brilho e Contraste Local

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

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

2.3.3 Geometria e Perspectiva

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

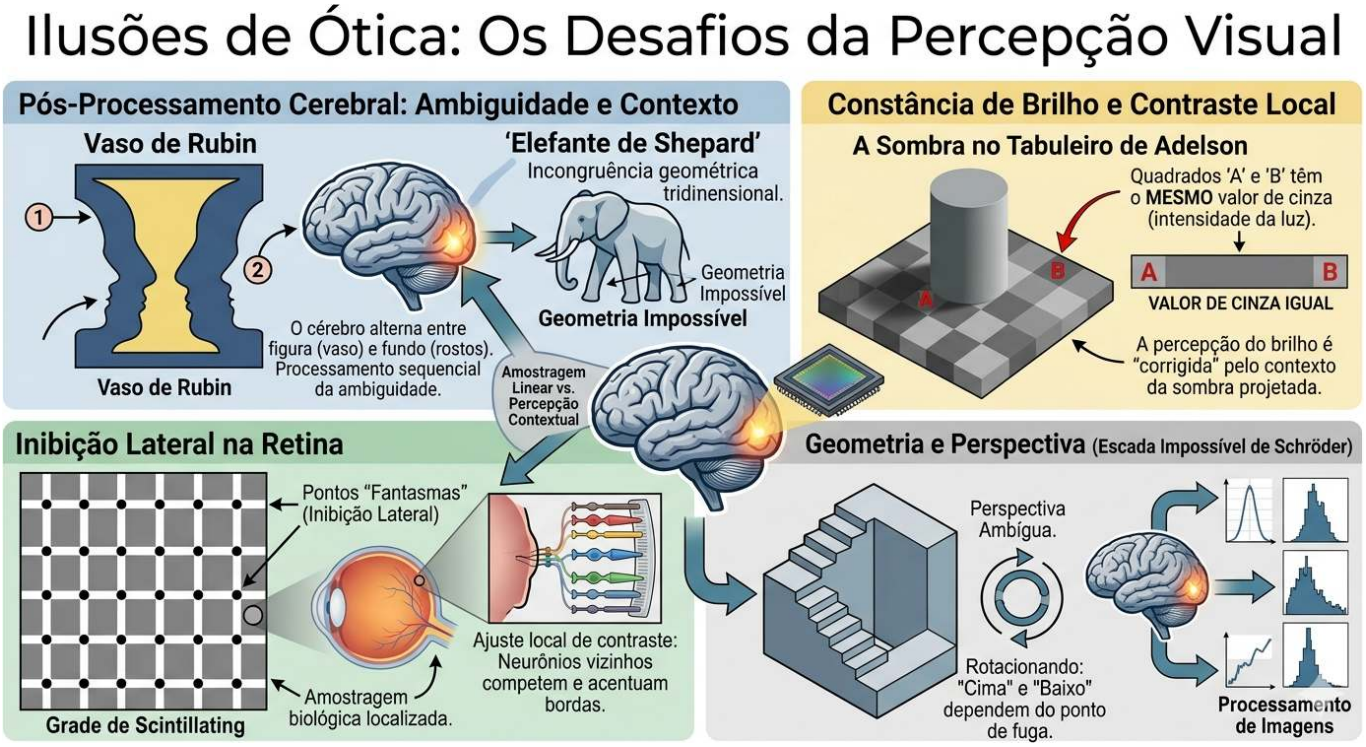


Figura 2.2: Coletânea de desafios perceptivos: (topo esquerdo) Vaso de Rubin — ambiguidade figura-fundo; (topo central) Elefante de Shepard — incongruência geométrica; (topo direita) Sombra de Adelson — constância de brilho baseada no contexto; (inferior esquerdo) Grade de pontos — inibição lateral;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

2.4 O Modelo Matemático da Formação da Imagem

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

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

onde:

- $i(x, y)$ é a **iluminação** incidente sobre a cena (energia luminosa por unidade de área), determinada pela fonte de luz;

- $r(x, y)$ é a **refletância** do objeto (fração da luz refletida), determinada pelas propriedades ópticas da superfície, com $0 < r(x, y) < 1$.

Na prática, as duas componentes variam em escalas espaciais distintas: $i(x, y)$ tende a variar lentamente no espaço, enquanto $r(x, y)$ pode apresentar variações abruptas associadas a texturas, bordas e detalhes finos. Técnicas de PDI frequentemente procuram separar ou compensar essas componentes, como em métodos de correção de iluminação não uniforme.

A Figura 2.3 ilustra como esse modelo se manifesta em imagens digitais coloridas, representadas por múltiplos canais espectrais (RGB) e, opcionalmente, por um canal adicional de transparência (RGBA).

2.4.1 Representação Digital e Canais Espectrais

Em imagens digitais coloridas, a função $f(x, y)$ da Equação 2.1 é representada por múltiplos canais espectrais. No padrão **RGB**, cada pixel armazena três amostras independentes:

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

correspondentes às intensidades das componentes vermelha (*Red*), verde (*Green*) e azul (*Blue*). Em imagens do tipo **RGBA**, adiciona-se um quarto canal:

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

onde $A(x, y)$ representa o canal **alfa** (*Alpha*), responsável por codificar a opacidade do pixel. Por convenção, o valor $A = 0$ indica um pixel totalmente transparente, enquanto $A = 255$ (ou 1) representa um pixel totalmente opaco.

Assim, uma imagem digital colorida é estruturada como uma matriz multidimensional de dimensões:

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

em que cada posição (x, y) armazena as amostras associadas às propriedades ópticas daquela coordenada espacial.

A conversão entre faixas de intensidade é direta e dada por:

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

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

Convenção de representação no `morph.hpp`: a biblioteca deste livro adota uma convenção única (Tabela 2.1), sem as ambiguidades de faixa e ordem de canais que surgem ao combinar várias bibliotecas Python.

Tabela 2.1: Convenção de representação de imagens em morph.hpp — faixa, ordem de bandas, nº de canais e layout de memória.

Aspecto	`morph.hpp`
Tipo de amostra	`unsigned char` (`uint8`), faixa $[0, 255]$
Ordem das bandas	**RGB** (via `stb_image`; sem inversão BGR)
Nº de canais	`1` (tons de cinza) ou `3` (RGB)
Layout na memória	$H \times W \times C$ intercalado (`data[(y*w + x)*channels + c]`)

A `struct Image` guarda os pixels num `std::vector<unsigned char>` linear, com os campos `h`, `w` e `channels`. A função `mm::read` decodifica com `stb_image` forçando 3 canais (ou 1, quando `grayscale=true`); portanto **um eventual canal alfa do arquivo é descartado na leitura**. Para trabalhar em ponto flutuante, vale a mesma relação $f_{\text{norm}} = f/255$.

Na célula de exemplo a seguir, a versão RGBA ($M \times N \times 4$) é montada empilhando um canal alfa sintético sobre o array lido — o kernel do notebook é Python, e a exibição recebe o `ndarray` nesse layout $H \times W \times C$.

2.4.2 Preparando o Ambiente Prático

O bloco a seguir carrega o módulo `morph.py` do repositório e demonstra, para um pixel sintético, as diferentes convenções de escala e ordem de canais adotadas pelas principais bibliotecas de PDI.

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # artefatos de build da trilha C++ (.cpp, binário,
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")

# O kernel é Python mesmo na trilha C++: `mm` (morph.py) é usado pelos
# simuladores, pela exibição das figuras que o binário C++ gera e pelo
# estado mm::Image entre células. cpp=True baixa também a trilha compilada
# (morph.hpp + stb_image*.h), usada no #include das células %%writefile *.cpp.
import config
config.setup(demo=True, cpp=True)
from morph import mm
```

```
Ambiente pronto. Morph: 1.1.9 | OpenCV: 5.0.0

Convenções de escala por biblioteca
NumPy / Pillow / OpenCV (uint8): [200 100  50] → [0, 255]
scikit-image / PyTorch (float):  [0.78431373 0.39215686 0.19607843] → [0.0, 1.0]
OpenCV: atenção - lê em BGR:     [ 50 100 200] (canais invertidos)
```

2.4.3 Implementação da Leitura de Imagem em `morph.hpp`

O trecho a seguir exibe o código-fonte da função `mm::read`, permitindo verificar diretamente como a biblioteca `morph.hpp` implementa a leitura de imagens.

```
# A morph.hpp é header-only; abaixo, o corpo da função 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 e RGBA: Exemplo Prático com a Imagem Mandrill

O exemplo a seguir carrega a imagem Mandrill em formato RGB, constrói artificialmente um canal alfa com gradiente horizontal e exibe ambas as representações, ilustrando concretamente as estruturas matriciais $M \times N \times 3$ e $M \times N \times 4$.

```
%%writefile tmp/fig_02_rgb_rgba.cpp
#define MM_OUT "tmp/fig_02_rgb_rgba.png"
#include <iostream>
#include <string>
#include "morph.hpp"
#include <filesystem>

int main() {
    // https://commons.wikimedia.org/wiki/File:Mandrill-k-means.png
    std::string url     =
"https://upload.wikimedia.org/wikipedia/commons/a/ab/Mandrill-k-means.png";
    std::string caminho = "imagens/mandrill.png";

    // Verifica se o arquivo existe (simulação: tenta ler; se falhar, baixa)
    // Nota: não há verificação direta de existência em C++, então tentamos ler
    try {
        mm::Image img_rgb_test = mm::read(caminho);
        // Se chegou aqui, o arquivo existe
    } catch (...) {
        // Cria diretório e baixa a imagem
        system("mkdir -p imagens");
        mm::write(mm::read(url), caminho);
    }

    mm::Image img_rgb = mm::read(caminho);  // (M, N, 3), uint8

    // Canal alfa: gradiente horizontal 0 -> 255 (esquerda -> direita)
    int h = img_rgb.h;
    int w = img_rgb.w;
    mm::Image alpha(h, w);  // zeros por padrão

    for (int x = 0; x < w; x++) {
        for (int y = 0; y < h; y++) {
            alpha.at(y, x) = static_cast<unsigned char>(255 * 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 k = 0; k < 3; k++) {
                img_rgba.at(y, x, k) = img_rgb.at(y, x, k);
            }
            img_rgba.at(y, x, 3) = alpha.at(y, x);
        }
    }

    std::cout << "RGB   -> shape=" << img_rgb.h << "x" << img_rgb.w << "x" << img_rgb.channels
<< std::endl;
    std::cout << "RGBA  -> shape=" << img_rgba.h << "x" << img_rgba.w << "x" <<
img_rgba.channels << std::endl;

    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=512x512x3
RGBA  -> shape=512x512x4
[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)")
```

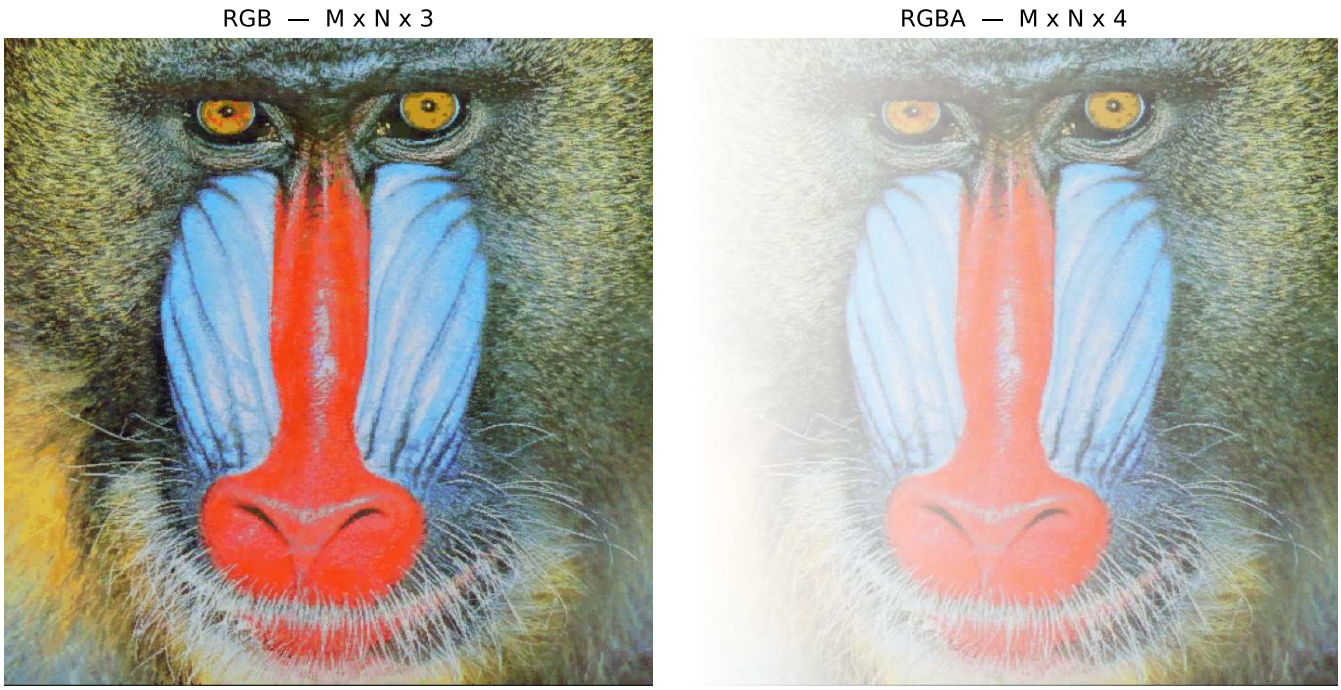


Figura 2.3: Imagem RGB (3 canais, $M \times N \times 3$) e versão RGBA (4 canais, $M \times N \times 4$) com transparência alfa gradual da esquerda para a direita. Imagem Mandrill — USC SIPI Image Database (domínio público).

2.5 Digitalização: Amostragem e Quantização

Para transformar uma cena contínua em uma imagem digital, dois processos são necessários: amostragem e quantização.

2.5.1 Amostragem - Discretização do espaço

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

2.5.2 Quantização - Discretização da intensidade

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

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

⚠ Erro de quantização

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

2.5.3 Efeitos da amostragem e quantização

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

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

int main() {
    // Imagem de exemplo (barbudo-rajado) - base das figuras de amostragem,
    // quantização e transformações geométricas deste capítulo.
    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";

    if (!std::filesystem::exists(caminho)) {
        std::filesystem::create_directories("imagens");
        mm::write(mm::read(url), caminho);
    }

    mm::Image img_color = mm::read(caminho);
    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 << std::endl;

    // As variáveis img_gray e img_color permanecem no escopo até o final da main()

// [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
```

O experimento apresentado na Figura 2.4 ilustra o compromisso entre a **resolução espacial** e o **custo de armazenamento** em memória. O código utiliza a técnica de subamostragem por fatiamento (*slicing*) para reduzir a matriz original de pixels conforme um fator f, resultando em uma economia de memória — por exemplo, um fator $f = 8$ reduz o tamanho do dado em 64 vezes (8^2). Para fins de comparação visual, as imagens reduzidas são restauradas às dimensões originais (512×512) através da interpolação por **vizinho mais próximo** (`nearest`). Este processo não recupera a informação perdida, mas torna evidente o efeito de ***aliasing*** e a estrutura de blocos (pixelização) gerada pela baixa densidade de dados da matriz amostrada.

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

```cpp
#include <utility>

// Função que retorna par (imagem, rótulo)
std::pair<mm::Image, std::string> subsample_simple(const mm::Image& image, int f) {
    // Subamostragem via fatiamento (slicing)
    mm::Image reduced = mm::subsample(image, f);

    // Cálculo de memória em 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((int)mem_kb) + " KB\n(Fator " +
                        std::to_string(f) + ")";

    // Restaura o tamanho para visualização (H, W originais)
    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]

    // img_gray já está disponível (inserido automaticamente)

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

    // Gera os resultados e separa em listas para o mm.show
    std::vector<mm::Image> imgs_list;
    std::vector<std::string> titles_list;

    for (int f : factors) {
        auto result = subsample_simple(img_gray, f);
        imgs_list.push_back(result.first);
        titles_list.push_back(result.second);
    }

    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
(Fator 1)
[2] 165x258, 41 KB
(Fator 4)
[3] 83x129, 10 KB
(Fator 8)
[4] 55x86, 4 KB
(Fator 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)")
```

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

O experimento na Figura 2.5 foca na **quantização de intensidade**, o processo de discretização da amplitude da função $f(x, y)$. Enquanto a subamostragem afeta a grade espacial, a redução da profundidade de bits limita a quantidade de tons de cinza disponíveis para representar o brilho.

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

```
%%writefile tmp/fig_02_quantizacao.cpp
#define MM_OUT "tmp/fig_02_quantizacao.png"
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <cmath>

// Função que reduz a profundidade de bits e calcula metadados de memória
std::pair<mm::Image, std::string> quantize_simple(const mm::Image& image, int bits) {
    int levels = std::pow(2, bits);

    // Normalização e quantização uniforme
    mm::Image quantized(image.h, image.w);
    for (int y = 0; y < image.h; y++) {
        for (int x = 0; x < image.w; x++) {
            double value = std::floor(image.at(y, x) / 256.0 * levels) / levels * 255.0;
            quantized.at(y, x) = static_cast<unsigned char>(value);
        }
    }

    // Cálculo de memória em KB
    double mem_kb = (quantized.h * quantized.w * quantized.channels) / 1024.0;
    std::string label = std::to_string(bits) + " bits (" + std::to_string(levels) + "
níveis)\n" +
                        std::to_string(static_cast<int>(mem_kb)) + " KB";

    return {quantized, 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]

    // Lista de bits para teste (8 é o padrão, 1 é o binário)
    std::vector<int> bits_test = {8, 4, 2, 1};

    // Gera os resultados e separa em listas para o mm.show
    std::vector<mm::Image> imgs_q;
    std::vector<std::string> titles_q;

    for (int b : bits_test) {
        auto result = quantize_simple(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)")
```

Figura 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 Análise Técnica

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

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

Embora linguagens como ANSI C permitam a compactação de 8 pixels por byte (*bit-packing*), tal abordagem exige a descompactação constante para cálculos e impõe alta complexidade na manipulação de ponteiros. Conforme a Tabela 2.2, privilegia-se o uso de `uint8` pela facilidade no acesso a vizinhos e pela versatilidade em transformações geométricas. Além disso, métodos nativos do NumPy e OpenCV executam o processamento internamente em baixo nível (C/C++), o que torna operações vetorizadas mais rápidas do que implementações manuais com laços aninhados em Python.

Tabela 2.2: Comparativo entre estratégias de compactação e eficiência de processamento.

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

2.6 Relações entre Pixels - Topologia da Imagem

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

2.6.1 Vizinhança

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

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

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

```
%%writefile tmp/fig_02_vizinhanca.cpp
#define MM_OUT "tmp/fig_02_vizinhanca.png"
// Compile: g++ -std=c++17 -o programa programa.cpp -I. -lstdc++fs `pkg-config --cflags --libs
opencv4` 2>/dev/null || g++ -std=c++17 -o programa programa.cpp -I.
// (se o morph.hpp estiver no diretório atual, o segundo comando funciona sem OpenCV)

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

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

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

    // ou simplesmente (teste com números como argumentos):
    mm::Image viz = mm::secross();

    // Exibição da matriz para análise de coordenadas
    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)")
```

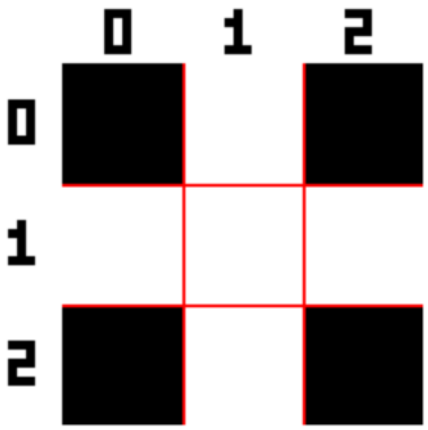


Figura 2.6: Ilustração de vizinhanças 4 em uma matriz 3x3. No centro (1,1), o pixel de interesse.

2.6.2 Adjacência, conectividade e caminhos

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

A **conectividade-4** (N4) e **conectividade-8** (N8) podem produzir resultados diferentes na segmentação e no cálculo de componentes conexas (etiquetagem). Por exemplo, um padrão de tabuleiro de xadrez pode ser completamente desconexo em N4, mas totalmente conexo em N8.

2.6.3 Distâncias entre pixels

As métricas de distância são fundamentais para quantificar a proximidade física e a conectividade entre os elementos que compõem a grade digital. Conforme demonstrado na Tabela 2.3, a escolha da métrica define o custo de deslocamento entre pixels e altera o comportamento de algoritmos de segmentação e análise morfológica.

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

Tabela 2.3: Comparativo de métricas de distância aplicadas à malha de pixels.

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

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

i Exemplo prático

Considerando dois pixels com deslocamentos relativos $\Delta x = 3$ e $\Delta y = 4$:

- **Euclidiana**: $\sqrt{3^2 + 4^2} = 5$ (hipotenusa do triângulo retângulo).
- ***Manhattan***: $3 + 4 = 7$ (soma dos catetos).
- ***Chebyshev***: $\max(3, 4) = 4$ (predomínio do maior deslocamento).

2.7 Armazenamento de Imagens

A escolha do formato de arquivo é um passo decisivo no fluxo de processamento, pois determina como os dados de amostragem e quantização serão preservados ou descartados. Conforme se apresenta na Tabela 2.4, cada extensão equilibra de forma distinta a fidelidade dos dados e a eficiência de armazenamento.

Tabela 2.4: Principais formatos de armazenamento de imagens digitais e suas aplicações em PDI.

Formato	Características	Uso típico
PGM	Formato simples de mapa de cinzas (texto ou binário).	Pesquisa acadêmica e ferramentas Unix.
BMP	Não comprimido (ou compressão simples).	Windows, aplicações legado.
PNG	Compressão **sem perdas** (*lossless*).	Web, imagens com transparência.
JPEG	Compressão **com perdas** (*lossy*), ideal para fotografias.	Fotos, câmeras digitais.
TIFF	Suporta múltiplas camadas e compressão variada.	Editoração, arquivamento.
RAW	Dados brutos do sensor, sem processamento.	Fotografia profissional.
DICOM	Padrão médico com metadados clínicos embutidos (paciente, equipamento, protocolo).	Radiologia, tomografia, ressonância magnética.

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

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

2.7.1 Exemplo: Extração de Metadados e Localização GPS

Diferente da matriz de pixels pura obtida pela leitura convencional — como na imagem do pássaro apresentada no início deste capítulo —, o uso do argumento `pil=True` no método `mm::read()` altera a natureza do objeto retornado (ver Figura 2.7). Enquanto o padrão (`pil=False`) retorna um `numpy.ndarray` RGB, a leitura com `pil=True` retorna um objeto especializado da biblioteca Pillow, capaz de interpretar o cabeçalho **EXIF**.

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

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

i Trilha C++: por que a extração de EXIF permanece em Python

A extração de metadados é **parsing de container** — decodificar o cabeçalho EXIF embutido no arquivo — e não processamento de pixels: é ortogonal ao pipeline de amostragem e quantização. A `morph.hpp` decodifica imagens com `stb_image`, que entrega apenas a matriz de pixels e descarta o cabeçalho EXIF. Como o kernel deste notebook é Python mesmo na trilha C++, este exemplo usa a leitura em modo Pillow (`pil=True`) nas duas trilhas. Em um programa C++ autônomo, o caminho equivalente seria vendorizar uma biblioteca dedicada: `easyexif` (leitura de EXIF/GPS em JPEG, header único, zero dependências) ou `exiv2` (leitura **e** escrita, incluindo IPTC/XMP).

```
%%writefile tmp/fig_01_natureza.cpp
#define MM_OUT "tmp/fig_01_natureza.png"
// Compile: g++ -std=c++17 -o programa programa.cpp -lmorph
#include "morph.hpp"
#include <iostream>
#include <fstream>
#include <string>
#include <cmath>
#include <sys/stat.h>
#include <sys/types.h>
#include <filesystem>

// Estrutura para armazenar dados EXIF básicos
struct GPSInfo {
    double lat;
    double lon;
    bool hasGPS;
};
```

```cpp
// Função para extrair dados GPS (simplificada - em C++ não temos acesso ao EXIF via mm)
GPSInfo extractGPS(const mm::Image& img) {
    // Nota: A biblioteca mm não fornece acesso ao EXIF
    // Esta é uma implementação simplificada que retorna dados padrão
    GPSInfo info;
    info.hasGPS = false;
    info.lat = 0.0;
    info.lon = 0.0;
    return info;
}

int main() {
    // Definir constantes
    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. Leitura - preserva objeto PIL com 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);  // salva preservando EXIF
    } else {
        img_obj = mm::read(caminho);
    }

    // Conversão para formato NumPy equivalente
    mm::Image img_numpy = img_obj;

    // 2. Extração e conversão de GPS (Tag 34853)
    GPSInfo gps = extractGPS(img_obj);
    if (gps.hasGPS) {
        std::cout << "GPS Decimal: " << gps.lat << ", " << gps.lon << "\n";
        std::cout << "Maps: https://www.google.com/maps/search/?api=1&query="
                  << gps.lat << "," << gps.lon << "\n";
    }

    // 3. Diagnóstico de tipos, dimensões e acesso a pixels
    std::cout << "\nTipo PIL  : mm::Image | Dimensões (x,y): "
              << img_obj.w << " x " << img_obj.h << "\n";

    // Acessar pixel (0,0) - RGB
    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) << ")\n";

    std::cout << "Tipo NumPy: mm::Image | Dimensões [y,x,c]: "
              << img_numpy.h << " x " << img_numpy.w << " x "
              << img_numpy.channels << "\n";

    std::cout << "NumPy [0,0]: ("
              << (int)img_numpy.at(0, 0, 0) << ", "
              << (int)img_numpy.at(0, 0, 1) << ", "
              << (int)img_numpy.at(0, 0, 2) << ")\n";
```

```
    // 4. Exibição
    mm::show(img_numpy, 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  : mm::Image | Dimensões (x,y): 2047 x 3067
Pillow (0,0): (58, 96, 0)
Tipo NumPy: mm::Image | Dimensões [y,x,c]: 3067 x 2047 x 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)")
```

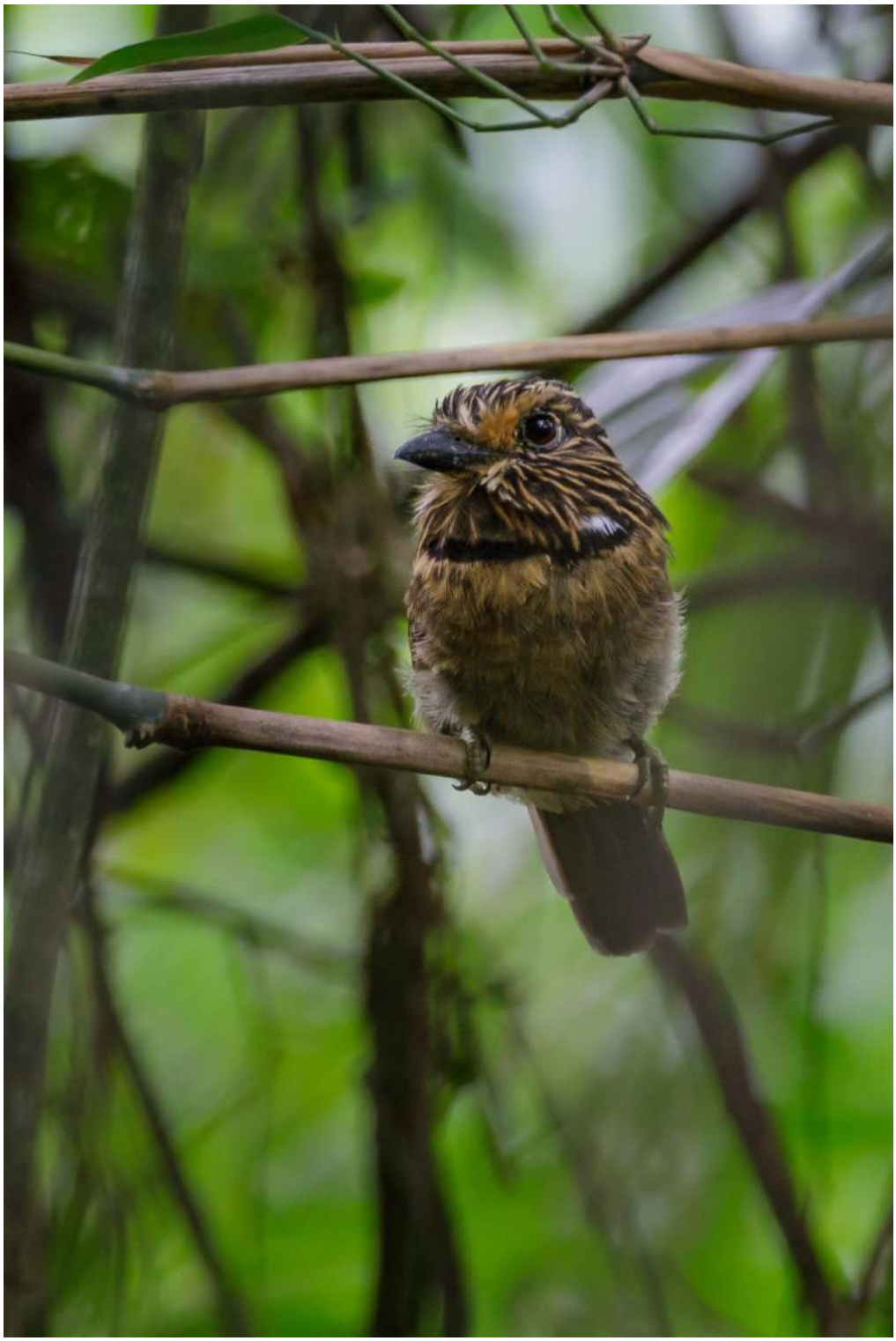


Figura 2.7: Area de Proteção Ambiental Quilombos do Médio Ribeira - Barbudo-rajado (Malacoptila striata). Crédito: Thomas Fuhrmann (CC BY-SA 4.0).

i Nota Pedagógica: A sutil diferença das dimensões

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

- **No Pillow (`.size`):** Retorna `(Largura, Altura)` — no exemplo: `(2047, 3067)`. É uma visão orientada ao arquivo de imagem.
- **No NumPy (`.shape`):** Segue a convenção matemática de matrizes: `(Linhas/Altura,`

`Colunas/Largura, Canais)` — no exemplo: `(3067, 2047, 3)`.
Esta distinção é fundamental para evitar erros de indexação ao implementar filtros manuais. Enquanto o objeto Pillow carrega o **"onde"** e o **"quando"** (contexto), o array NumPy carrega o **"quanto"** de luz (intensidade) existe em cada ponto da imagem.

2.7.2 Por que esta separação é importante?

Ao carregar uma imagem pelo caminho convencional (`pil=False`), o resultado é um `numpy.ndarray`, que contém estritamente os valores numéricos resultantes da **quantização** e **amostragem**. No entanto, ao utilizar `pil=True`, o `mm::read()` retorna um objeto da classe `PIL.JpegImagePlugin.JpegImageFile`.

Esta classe mantém o arquivo "aberto" para permitir o acesso ao contexto da captura antes que os dados sejam convertidos em uma matriz bruta. Note que o pixel `(0, 0)` é idêntico nas duas representações — `(58, 96, 0)` no Pillow e `[58, 96, 0]` no NumPy —, confirmando que ambos descrevem os mesmos dados, apenas com interfaces distintas. Essa separação é vital: pixels servem para algoritmos; metadados servem para georreferenciamento, catalogação científica e correções baseadas no hardware de aquisição.

Para inspecionar todos os metadados EXIF de um objeto Pillow:

```
from PIL import ExifTags

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

2.8 Transformações Geométricas Básicas

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

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

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

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

i Transformação inversa e interpolação

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

2.8.1 Translação

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

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

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

Pixels deslocados para além da área original são descartados; áreas descobertas são preenchidas com 0 (preto). Ver um exemplo na Figura 2.8.

```
%%writefile tmp/fig_02_translacao.cpp
#define MM_OUT "tmp/fig_02_translacao.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_22.png");
// [pdi:state-io:end]

    // img_gray já está inicializado aqui (inserido automaticamente)

    // Exemplo de translação da imagem do pássaro com deslocamentos (50,50) e (100,50).
    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)")
```

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

2.8.2 Rotação

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

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

Na implementação, `cv2.getRotationMatrix2D` gera diretamente as duas primeiras linhas de $\mathbf{T}_{\text{rot}}$ (matriz 2×3 para `warpAffine`), aceitando também um fator de escala s que multiplica $\cos\theta$ e $\sin\theta$. Ver exemplo na Figura 2.9.

Como a rotação desloca pixels para novas posições não-inteiras, `cv2.warpAffine` precisa estimar a cor de

cada pixel de destino a partir dos vizinhos — processo chamado **interpolação**. O parâmetro `interp` controla esse comportamento:

- **`nearest`** (`INTER_NEAREST`): atribui o valor do pixel mais próximo. Rápido, mas produz serrilhado (*aliasing*) em bordas diagonais.
- **`bilinear`** (`INTER_LINEAR`, padrão): média ponderada dos 4 vizinhos mais próximos. Equilibra qualidade e desempenho — adequado para a maioria dos casos.
- **`bicubic`** (`INTER_CUBIC`): considera os 16 vizinhos em uma superfície cúbica. Produz bordas mais suaves, ao custo de maior processamento.

```
%%writefile tmp/fig_02_rotacao.cpp
#define MM_OUT "tmp/fig_02_rotacao.png"
#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]

    // Exemplo de rotação da imagem do pássaro em 30° e 45° usando interpolação bilinear.
    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)")
```

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

2.8.3 Escala (Redimensionamento)

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

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

Quando $s > 1$ (ampliação), pixels da imagem destino mapeiam para posições não inteiras na origem — exigindo interpolação para estimar o valor. Quando $s < 1$ (redução), múltiplos pixels de origem contribuem para um único pixel destino — exigindo **decimação**. Os mesmos três métodos descritos na rotação estão disponíveis em `mm::resize`, com a diferença de que aqui o impacto visual é mais perceptível: na ampliação, `nearest` produz efeito de blocos (*pixelation*), enquanto `bicubic` preserva melhor a nitidez das bordas, conforme a Tabela 2.5:

Tabela 2.5: Métodos de interpolação disponíveis em mm.resize e seus respectivos números de vizinhos utilizados no cálculo.

Método	Vizinhos usados	Característica
`'nearest'`	1	Rápido; produz efeito de blocos (*pixelation*)
`'bilinear'`	4	Bom compromisso qualidade/custo; bordas suaves

Método	Vizinhos usados	Característica
`'bicubic'`	16	Maior nitidez; preferido em softwares profissionais

O parâmetro `size_or_factor` aceita tanto um escalar (fator uniforme, e.g. `0.5` para reduzir à metade) quanto uma tupla `(largura, altura)` para dimensões absolutas.

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

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

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. Recorte da região do bico
    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. Ampliar 4× com mm.resize
    mm::Image crop_nearest  = mm::resize(crop, 4.0, "nearest");
    mm::Image crop_bilinear = mm::resize(crop, 4.0, "bilinear");

    // 3. Exibição comparativa
    mm::show(
        std::vector<mm::Image>{crop, crop_nearest, crop_bilinear},
        MM_OUT,
        std::vector<std::string>{"Original (recorte)", "Vizinho mais próximo (4×)", "Bilinear
(4×)"},
        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 (4×)
[3] Bilinear (4×)
```

```
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)")
```

Figura 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 Cisalhamento (*Shear*)

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

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

Para $sh_x \neq 0$ e $sh_y = 0$, cada linha é deslocada horizontalmente proporcionalmente à sua posição vertical — produzindo o efeito de "inclinação" característico. Ver exemplo na Figura 2.11.

```
%%writefile tmp/fig_02_cisalhamento.cpp
#define MM_OUT "tmp/fig_02_cisalhamento.png"
```

```
#include <iostream>
#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_22.png");
// [pdi:state-io:end]

    // img_gray já está inicializado (inserido automaticamente)

    // Exemplo de cisalhamento da imagem do pássaro: horizontal (shx=0.3),
    // vertical (shy=0.3) e combinado (shx=0.2, shy=0.2)
    mm::Image img_shx = mm::shear(img_gray, 0.3, 0.0);
    mm::Image img_shy = mm::shear(img_gray, 0.0, 0.3);
    mm::Image img_shc = mm::shear(img_gray, 0.2, 0.2);

    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)")
```

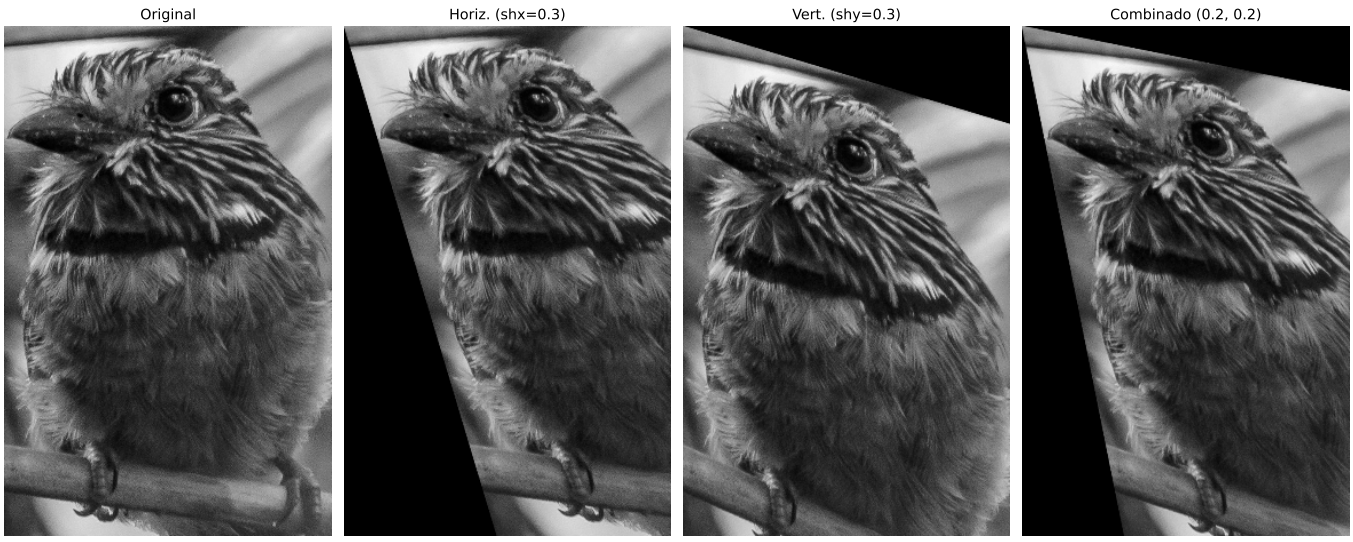


Figura 2.11: Exemplo de cisalhamento da imagem do pássaro: horizontal (shx=0.3), vertical (shy=0.3) e combinado (shx=0.2, shy=0.2).

2.9 Resumo

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

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

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

2.10 🤖 Uso do Gemini Notebook como Tutor Complementar

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

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

🎓 Estude com o Tutor Inteligente

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

🌐 Idioma e Linguagem de Programação

O projeto deste capítulo no Gemini Notebook foi construído apenas com o texto em **português** e os exemplos de código em **Python**. Se você está estudando pela edição em inglês ou francês, ou acompanhando a trilha em C++, as respostas do tutor podem não corresponder exatamente à versão que você está lendo.

⚠️ Aviso sobre Conteúdo Gerado por IA

A IA é uma poderosa aliada nos estudos, mas o conteúdo gerado pode conter **erros ou imprecisões**. Consulte sempre **livros, artigos científicos e outras fontes acadêmicas confiáveis** para validar as informações. Sempre que possível, execute os exemplos práticos fornecidos neste capítulo para verificar os resultados.

2.11 Lista de Exercícios

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

Referências do Capítulo

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

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

2.12 💻 Parte Prática com Exercícios de Programação

🎯 Objetivo deste Caderno

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

Download

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

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # artefatos de build da trilha C++ (.cpp, binário,
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")

# O kernel é Python mesmo na trilha C++: `mm` (morph.py) é usado pelos
# simuladores, pela exibição das figuras que o binário C++ gera e pelo
# estado mm::Image entre células. cpp=True baixa também a trilha compilada
# (morph.hpp + stb_image*.h), usada no #include das células %%writefile *.cpp.
import config
config.setup(testsuite=True, cpp=True)
from morph import mm
from testsuite import TestSuite
```

```
Ambiente pronto. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Executando os Testes

Para avaliar os testes, execute `TestSuite("EP04_01.extensão").run()` numa nova célula, trocando a extensão pela da linguagem usada (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). O sistema baixa os casos de teste do GitHub, executa o programa e calcula a nota automaticamente.

Para testar código Python diretamente, sem salvar arquivo, use `run_code(codigo)` passando o código como *string* numa variável `codigo`:

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

2.12.1 EP02_01 ⊙ Ajuste de Brilho e Contraste

Nesta atividade, o objetivo é implementar um operador de ponto para **transformação linear de intensidade**, aplicando o ajuste dinâmico de brilho e contraste em uma imagem digital.

2.12.1.1 📋 Diretrizes de Implementação

O algoritmo deve seguir o fluxo de execução abaixo:

1. **Dimensões:** Ler os inteiros L (linhas) e C (colunas) da matriz.
2. **Parâmetros:** Ler o valor real α (fator de contraste) e o inteiro β (fator de brilho).
3. **Dados:** Ler os valores inteiros da matriz original.
4. **Mapeamento:** Para cada pixel p, calcular o novo valor p' pela equação:

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

5. **Saída:** Exibir a matriz resultante com as dimensões $L \times C$.

2.12.1.2 📌 Restrições Computacionais

- **Arredondamento (*Round*):** Aplica-se o arredondamento matemático para o inteiro mais próximo antes da conversão de tipo.
- **Saturação (*Clipping*):** Os valores devem ser confinados ao intervalo $[0, 255]$ para preservar o padrão de 8 bits:

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

- **Simulação:** O efeito dos parâmetros α e β na correção histogramática pode ser observado na Figura 2.12.

2.12.1.3 🧠 Fundamentação Teórica

As alterações modificam o histograma da imagem para ajustar o perfil de iluminação e a distinção tonal.

Parâmetro	Função	Impacto Visual
α (***Alpha***)	Escalar	Modula o **Contraste**. Se $\alpha > 1$, expande o histograma; se $0 \leq \alpha < 1$, comprime.
β (***Beta***)	Aditiva	Modula o **Brilho**. Se positivo, translada o histograma para a direita; se negativo, para a esquerda.
clip	Limitador	Restringe a faixa dinâmica, impedindo erros de *underflow* e *overflow*.

2.12.1.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linha 3: Valores de `alpha` (α) e `beta` (β).
- Linhas seguintes: Elementos numéricos da matriz original.

Saída:

- Matriz transformada estruturada em L linhas e C colunas.

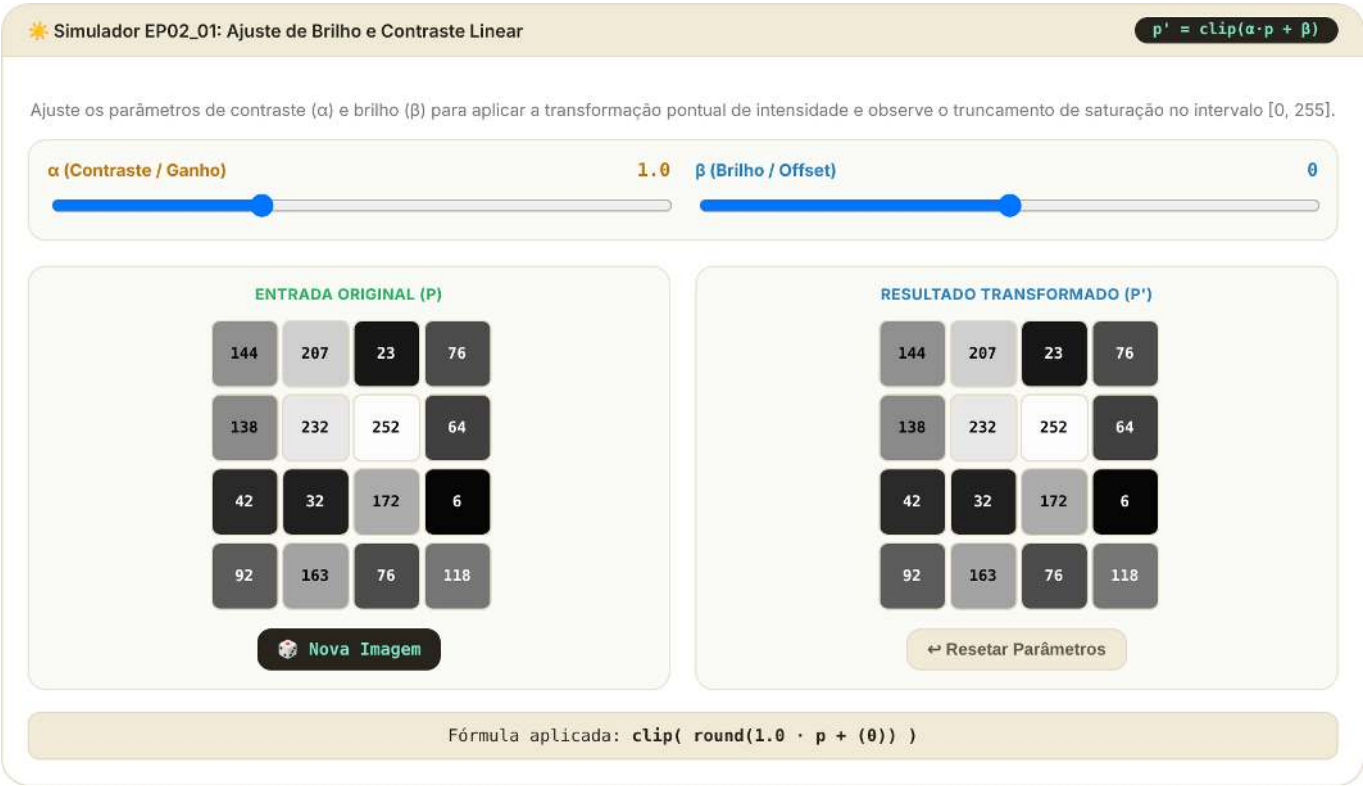
2.12.1.5 📌 Exemplos

A tabela a seguir apresenta um exemplo prático do comportamento esperado do algoritmo, destacando a atuação dos operadores de arredondamento e saturação.

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

Executando os Testes

Para avaliar os testes, execute `TestSuite("EP02_01.extensão").run()` numa nova célula, trocando a extensão pela da linguagem usada (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). O sistema baixa os casos de teste do GitHub, executa o programa e calcula a nota automaticamente.



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

Figura 2.12: Simulador EP02_01: Ajuste de Brilho e Contraste Linear (p' = p +)

```
%%writefile EP02_01.cpp
// sua solução
```

```
Overwriting EP02_01.cpp
```

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

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

2.12.2 EP02_02 🔬 Subamostragem Espacial

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

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz original.
- Leia um valor inteiro f ($f \geq 1$), que representa o fator de amostragem.

- Leia os valores inteiros da matriz original.
- A nova imagem deve ser construída selecionando o pixel da posição $(f \cdot i, f \cdot j)$ da imagem original.
- Imprima a matriz resultante com as novas dimensões.
- Ver na Figura 2.13 uma simulação deste EP.

📌 **Importante**:

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

2.12.2.1 🧠 Discretização do Espaço

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

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

2.12.2.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

A terceira linha contém o fator `f`.

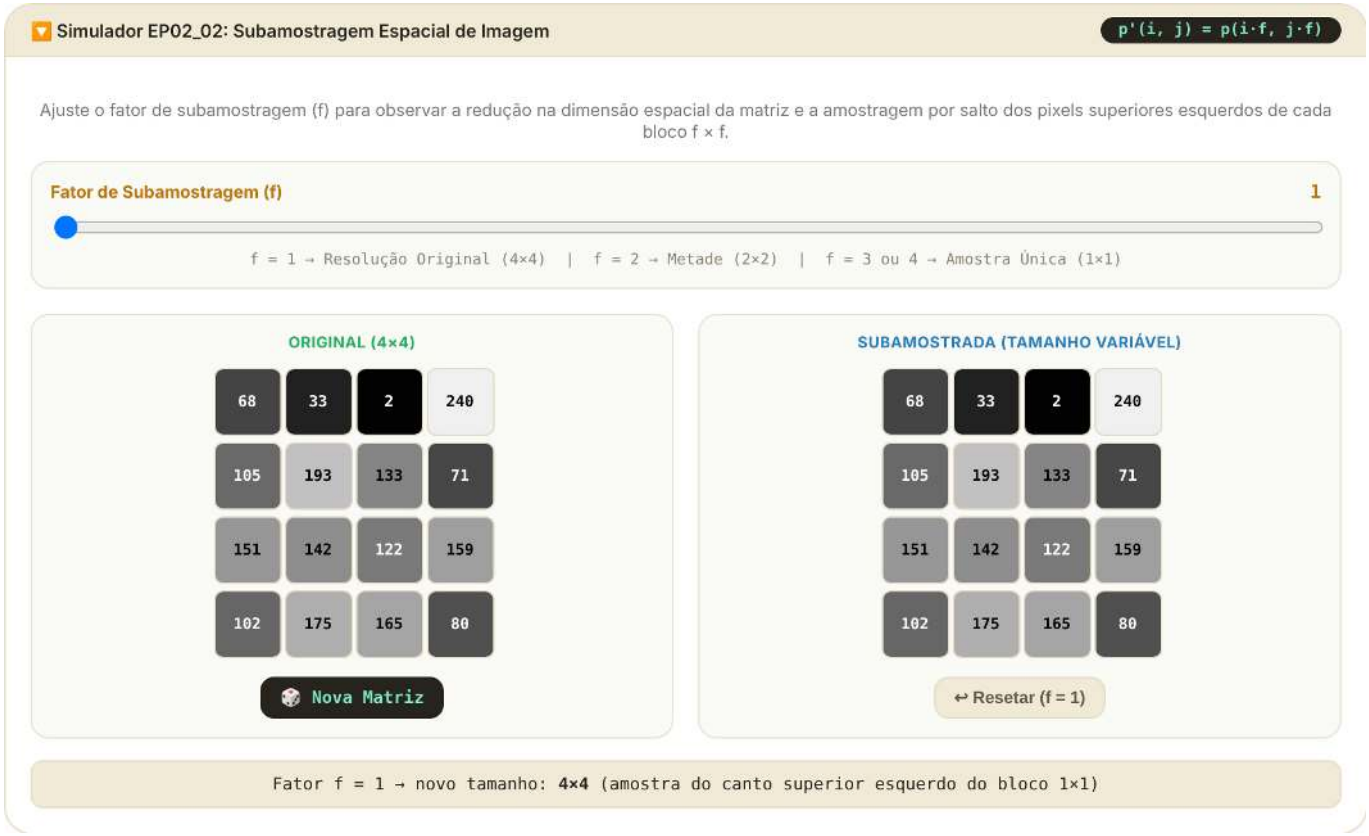
As linhas seguintes contêm os elementos da matriz $L \times C$.

Saída:

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

2.12.2.3 📌 Exemplos

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



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

Figura 2.13: Simulador EP02_02: Subamostragem Espacial (Redução de Resolução por Salto f)

```
%%writefile EP02_02.cpp
// sua solução
```

```
Overwriting EP02_02.cpp
```

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

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

2.12.3 EP02_03 🎨 Quantização de Níveis de Cinza

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

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz.
- Leia um inteiro k ($1 \leq k \leq 8$), representando o novo número de bits da imagem.
- Calcule o número de níveis ($N = 2^k$) e o tamanho do intervalo (passo).
- Para cada pixel p, calcule o novo valor p' mapeando-o para o índice do nível discretizado correspondente (variando de 0 a $2^k - 1$).
- Imprima a matriz resultante com os mesmos valores de dimensões originais.
- Ver na Figura 2.14 uma simulação deste EP.

📌 **Importante**:

- **Posterização:** Ao reduzir drasticamente os níveis (ex: $k = 2$), você notará que degradês suaves se transformam em faixas abruptas de cor devido à perda de resolução de amplitude.
- **Cálculo do Passo:** O intervalo entre cada nível é definido por $passo = 256/2^k$.

- **Mapeamento:** O método de quantização uniforme por truncamento que mapeia o pixel para o índice do seu respectivo nível discretizado é dado por:

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

Em termos de implementação (como em Python), isso equivale à divisão inteira: `p' = p // passo`.

2.12.3.1 🧠 Discretização da Amplitude

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

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

2.12.3.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

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

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

Saída:

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

2.12.3.3 📌 Exemplos

Entrada	Saída	Observação
1 4 2 0 80 170 255	0 1 2 3	Com $k = 2$, temos $2^2 = 4$ níveis discretos disponíveis $(0, 1, 2, 3)$. O passo é $256/4 = 64$. Aplicando a divisão inteira por elemento: $0//64 = 0$, $80//64 = 1$, $170//64 = 2$, $255//64 = 3$.
1 5 1 10 50 120 200 250	0 0 0 1 1	Com $k = 1$, temos $2^1 = 2$ níveis (0 e 1). Passo $= 256/2 = 128$. Pixels menores que 128 resultam em 0, e pixels maiores ou iguais a 128 resultam em 1.

```
%%writefile EP02_03.cpp
// sua solução
```

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

Figura 2.14: Simulador EP02_03: Quantização e Profundidade de Bits (Redução do Número de Níveis de Cinza)

```
Overwriting EP02_03.cpp
```

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

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

2.12.4 EP02_04 📐 Transformada de Distância em Imagem Binária

Dada uma imagem binária onde pixels de valor **1** representam o *objeto* e pixels **0** representam o *fundo*, a distância de um pixel de fundo recebe a **menor distância ao pixel de objeto mais próximo**. Pixels de objeto recebem distância **0**. Para simplificar, considere que a imagem tem **apenas um único objeto com um pixel de valor 1**.

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

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

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

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

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

2.12.4.1 🖼️ Por que isso importa? - Aplicações da DT

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

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

2.12.4.2 📌 Requisitos Técnicos

- **Entrada:** * Primeira linha: L e C (inteiros).
  - Segunda linha: nome da métrica (`euclidean`, `cityblock` ou `chessboard`).
  - Em seguida, a matriz binária $L \times C$ (valores 0 ou 1).
- **Pixels de objeto (1):** distância = 0 (ou 0.00 para euclidiana).
- **Pixels de fundo (0):** distância ao único pixel de objeto na imagem.
- **Arredondamento (euclidiana):** imprimir com **2 casas decimais** (formato `:.2f`). City-block e Chessboard produzem inteiros — imprimir sem decimais.
- **Saída:** valores separados por espaço, uma linha por linha da matriz.
- Ver na Figura 2.15 uma simulação deste EP.

2.12.4.3 📌 Exemplos

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

2.12.4.4 📌 Observações finais

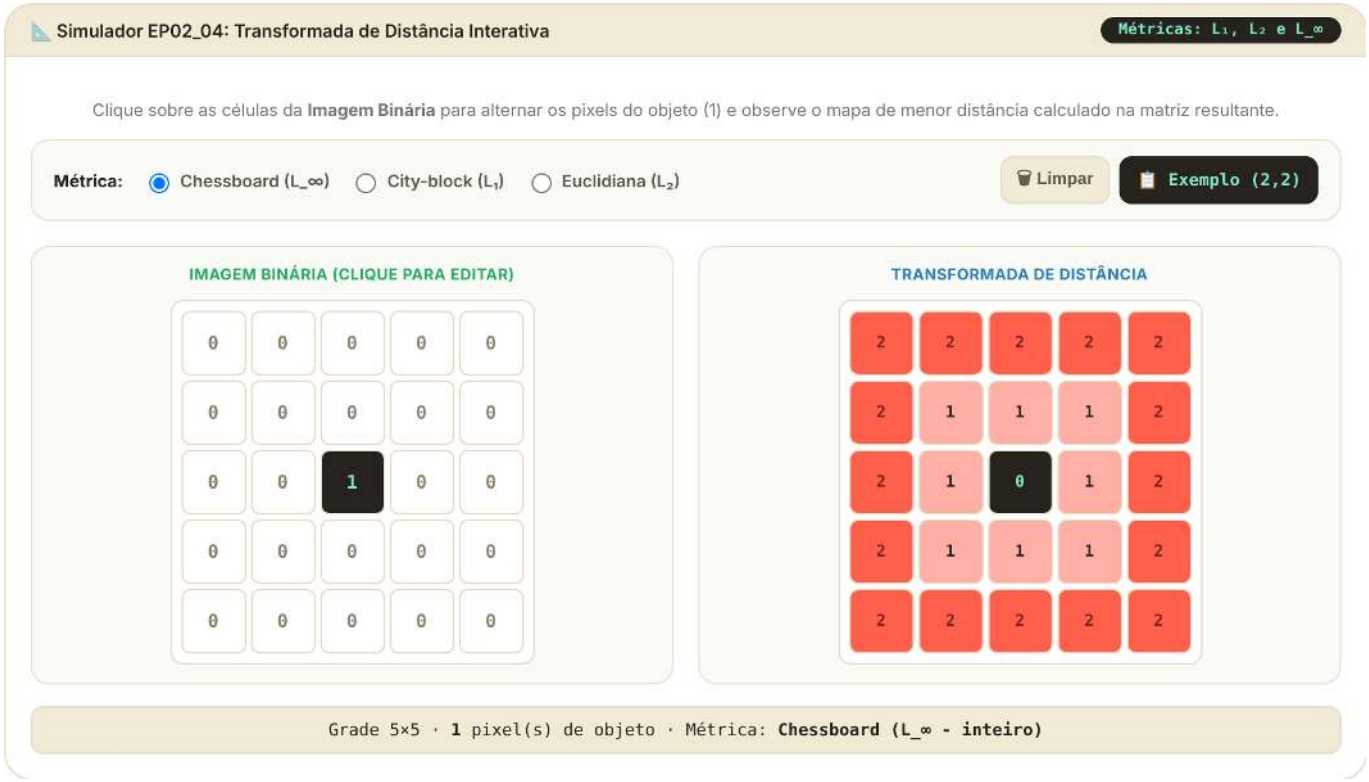
- Como a imagem tem **apenas um objeto de um pixel**, a distância de cada pixel de fundo é simplesmente a distância desse pixel ao único ponto objeto.
- A implementação pode usar força bruta (percorrer todos os pixels da imagem e calcular a distância diretamente), pois L e C são pequenos nos casos de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

```
%%writefile EP02_04.cpp
// sua solução
```

```
Overwriting EP02_04.cpp
```

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

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



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

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

2.12.5 EP02_05 Translação de Imagem

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

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz.
- Leia dois inteiros t_x (deslocamento horizontal) e t_y (deslocamento vertical).
- Leia os valores inteiros da matriz original.
- Calcule a nova posição (x', y') para cada pixel (x, y) original.
- Imprima a matriz resultante com as mesmas dimensões da original.
- Ver na Figura 2.16 uma simulação deste EP.

📌 **Importante**:

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

2.12.5.1 🧠 Deslocamento Espacial

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

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

Que resulta nas equações simples:

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

2.12.5.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

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

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

Saída:

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

2.12.5.3 📌 Exemplos

Entrada	Saída	Observação
2 2 1 1 10 20 30 40	0 0 0 10	**Deslocamento** $(t_x = 1, t_y = 1)$**:** Cada pixel move uma posição para a direita (horizontal) e uma para baixo (vertical). O pixel $(0, 0) = 10$ vai para o destino $(1, 1)$ (canto inferior direito). As posições vazias são preenchidas com 0.
3 3 -1 0 1 2 3 4 5 6 7 8 9	2 3 0 5 6 0 8 9 0	**Deslocamento** $(t_x = -1, t_y = 0)$**:** Cada pixel move uma posição para a esquerda (horizontal). A primeira coluna original (`1`, `4`, `7`) é descartada, as demais colunas movem-se para a esquerda, e a última coluna resultante é preenchida com zeros (0).

```
%%writefile EP02_05.cpp
// sua solução
```

```
Overwriting EP02_05.cpp
```

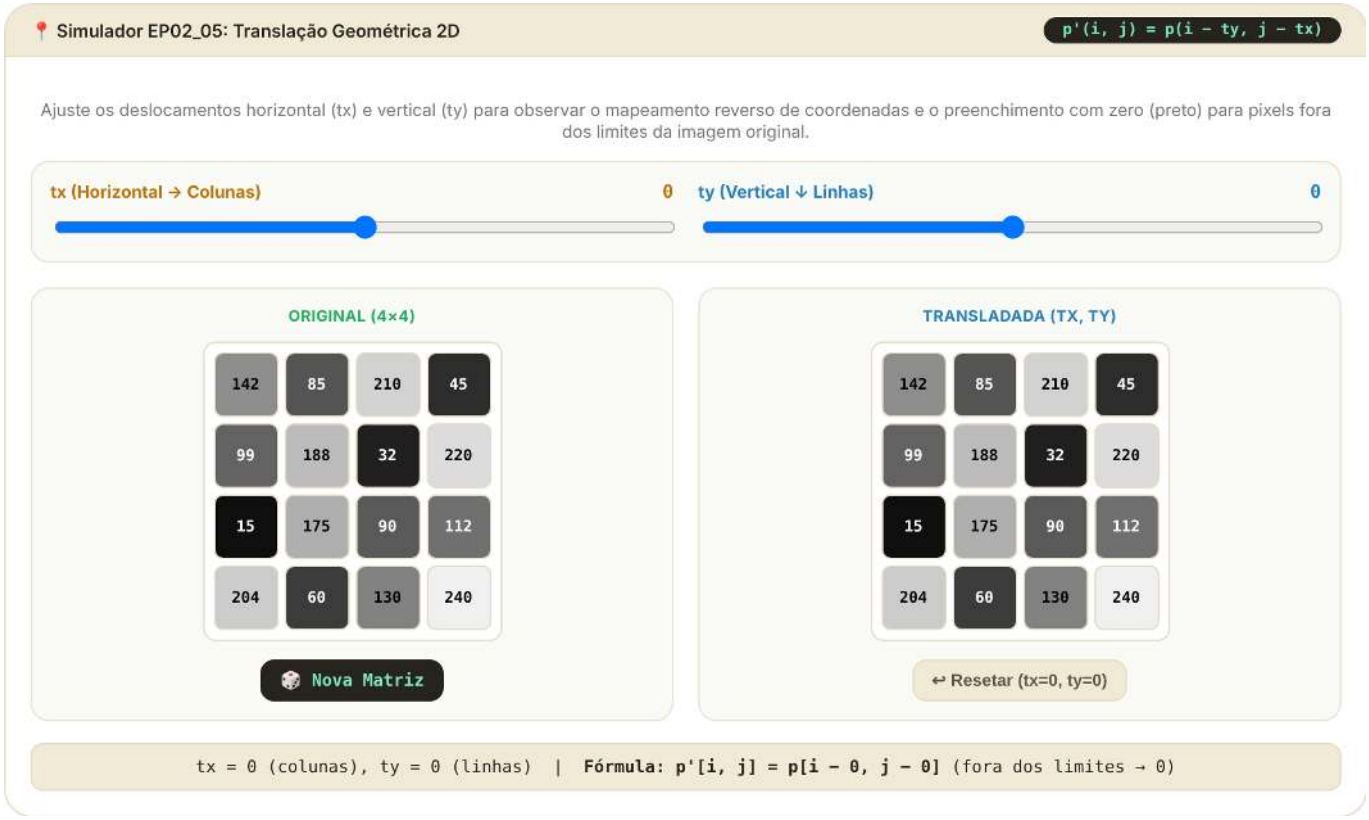
```
TestSuite("EP02_05.cpp").run()
```

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

2.12.6 EP02_06 🔄 Rotação de Imagem

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

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz.



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

Figura 2.16: Simulador EP02_05: Translação Geométrica de Imagem (Deslocamento tx e ty com Preenchimento de Borda)

- Leia um valor real θ (ângulo em graus) e uma *string* representando o **método** de interpolação (`nearest` ou `bilinear`).
- Leia os valores inteiros da matriz original.
- Realize a rotação em torno do centro da imagem $(L/2, C/2)$.
- Imprima a matriz resultante com as mesmas dimensões da original.
- Ver na Figura 2.17 uma simulação deste EP.

📌 **Importante**:

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

2.12.6.1 🧠 Transformação por Ângulo

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

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

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

2.12.6.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

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

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

Saída:

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

2.12.6.3 📌 Exemplos

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

```
%%writefile EP02_06.cpp
// sua solução
```

```
Overwriting EP02_06.cpp
```

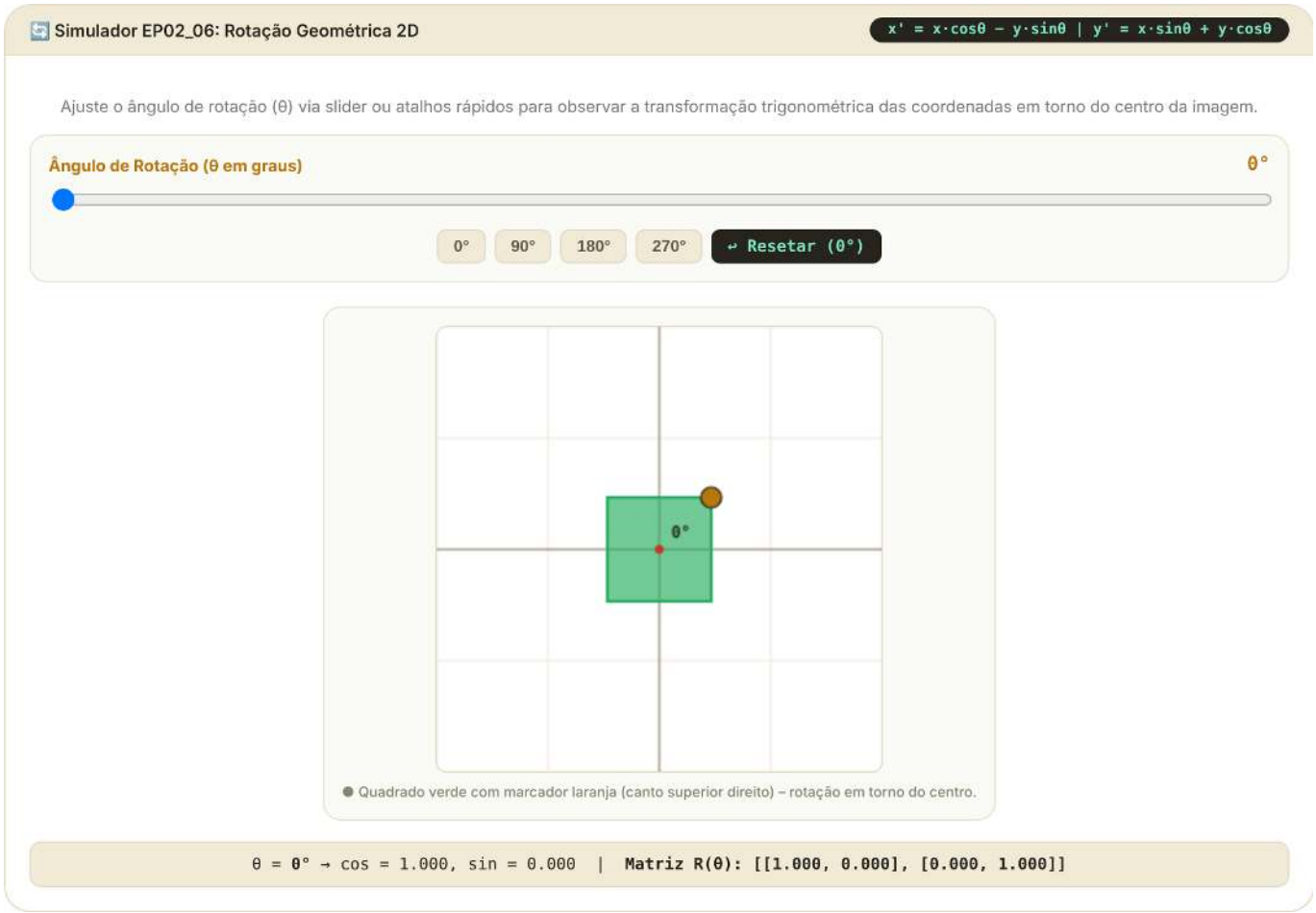
```
TestSuite("EP02_06.cpp").run()
```

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

2.12.7 EP02_07 🔍 Redimensionamento (Escala)

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

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz original.



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

Figura 2.17: Simulador EP02_06: Rotação de Imagem em Torno da Origem por Ângulo

- Leia dois valores reais s_x (escala nas linhas) e s_y (escala nas colunas).
- Leia uma *string* representando o **método** de interpolação (`nearest` ou `bilinear`).
- Leia os valores inteiros da matriz original.
- Calcule as novas dimensões: $L' = \text{round}(L \times s_x)$ e $C' = \text{round}(C \times s_y)$.
- Imprima a matriz resultante com as novas dimensões.
- Ver na Figura 2.18 uma simulação deste EP.

📌 **Importante**:

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

2.12.7.1 🧠 Interpolação para Ampliação/Redução

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

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

2.12.7.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

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

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

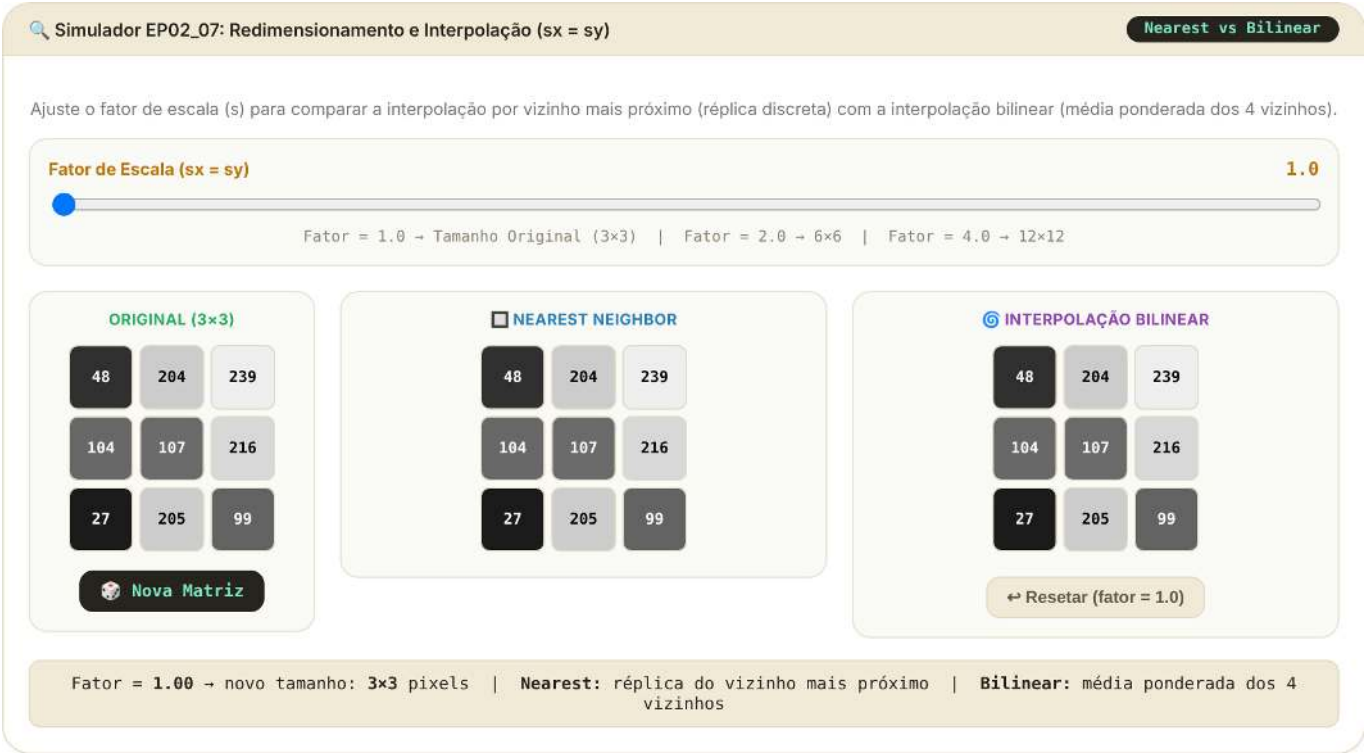
As linhas seguintes contêm os elementos da matriz $L \times C$.

Saída:

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

2.12.7.3 📌 Exemplos

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



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

Figura 2.18: Simulador EP02_07: Redimensionamento Espacial e Interpolação (Nearest Neighbor vs Bilinear)

```
# Ainda não portado para esta linguagem nesta versão - referência conceitual em Python.
%%writefile EP02_07.py
# Código Python
import numpy as np
from morph import mm

# 1. Leitura das dimensões, fatores e método
l = int(input())
c = int(input())
sx, sy = map(float, input().split())
interp = input().strip()

# 2. Leitura da imagem original
img = mm.readImg(l, c)
```

```
# 3. Novas dimensões
l_new = round(l * sx)
c_new = round(c * sy)

# 4. Redimensionamento usando mm.resize
# cv2.resize usa (largura, altura) = (colunas, linhas)
resultado = mm.resize(img, (c_new, l_new), method=interp)

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

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

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

2.12.8 EP02_08 🔀 Cisalhamento (Shear)

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

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz.
- Leia dois valores reais sh_x (cisalhamento horizontal) e sh_y (cisalhamento vertical).
- Leia uma *string* representando o **método** de interpolação (`nearest` ou `bilinear`).
- Leia os valores inteiros da matriz original.
- Aplique a transformação mantendo o tamanho original da imagem (cortando o que ultrapassar os limites).
- Imprima a matriz resultante com as dimensões $L \times C$.
- Ver na Figura 2.19 uma simulação deste EP.

📌 **Importante**:

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

2.12.8.1 🧠 Distorção Afim

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

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

Isso resulta nas equações:

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

2.12.8.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

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

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

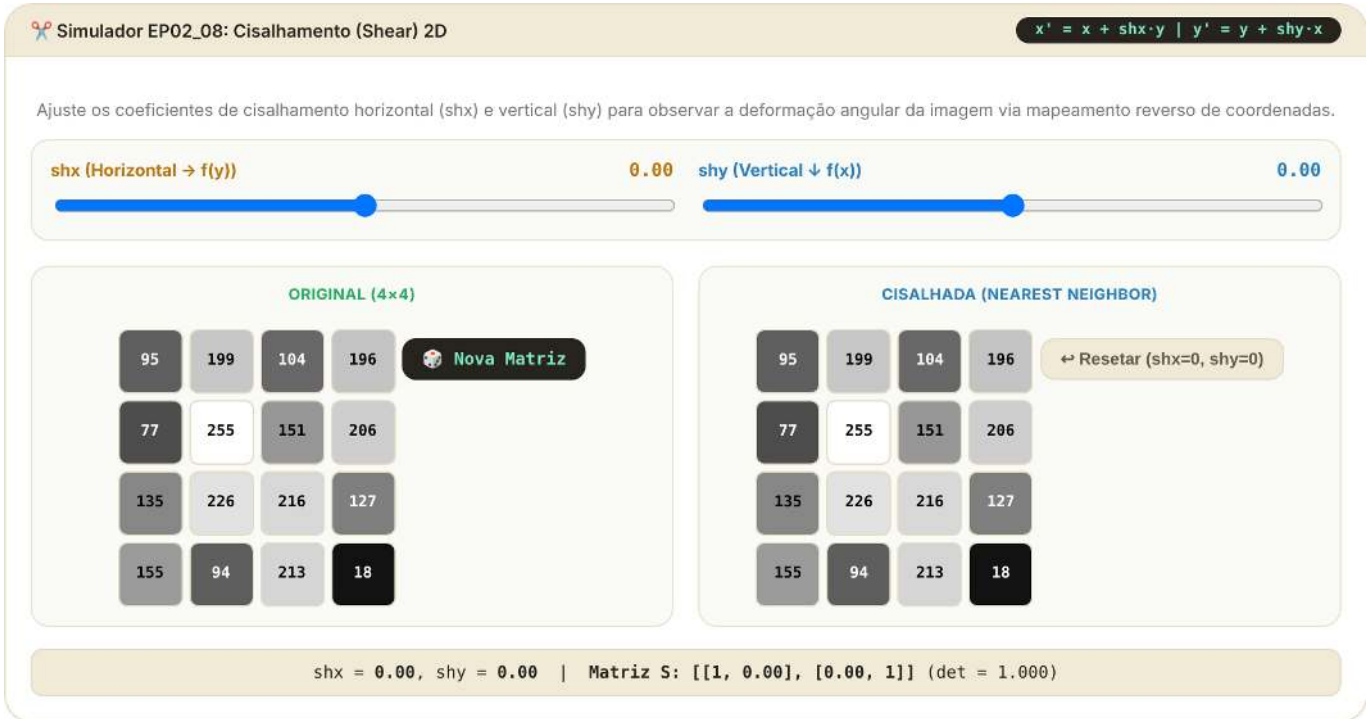
As linhas seguintes contêm os elementos da matriz $L \times C$.

Saída:

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

2.12.8.3 📌 Exemplos

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap02/sim-ep0208-cisalhamento.html

Figura 2.19: Simulador EP02_08: Transformaçao Geométrica de Cisalhamento 2D (Shear Horizontal e Vertical)

```
%%writefile EP02_08.cpp
// sua solução
```

```
Overwriting EP02_08.cpp
```

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

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

2.12.9 EP02_09 🧩 Transformação Afim Genérica

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

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz.
- Leia **seis valores reais** (a, b, t_x, c, d, t_y) que compõem a matriz de transformação afim 2×3.
- Leia uma *string* representando o **método** de interpolação (`nearest` ou `bilinear`).
- Leia os valores inteiros da matriz original.
- Aplique a transformação mantendo o tamanho original $L \times C$.
- Imprima a matriz resultante.
- Ver na Figura 2.20 uma simulação deste EP.

📌 **Importante**:

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

Dica:

```
flags = cv2.INTER_NEAREST if interp == 'nearest' else \
        cv2.INTER_CUBIC   if interp == 'bicubic'  else \
        cv2.INTER_LANCZOS4 if interp == 'lanczos' else \
        cv2.INTER_LINEAR

r = cv2.warpAffine(img, M, (C, L), flags=flags)
```

2.12.9.1 🧠 Combinação de Operações

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

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

Ou, de forma compacta em coordenadas homogêneas:

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

2.12.9.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

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

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

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

Saída:

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

2.12.9.3 📌 Exemplos

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

```
%%writefile EP02_09.cpp
// sua solução
```

```
Overwriting EP02_09.cpp
```

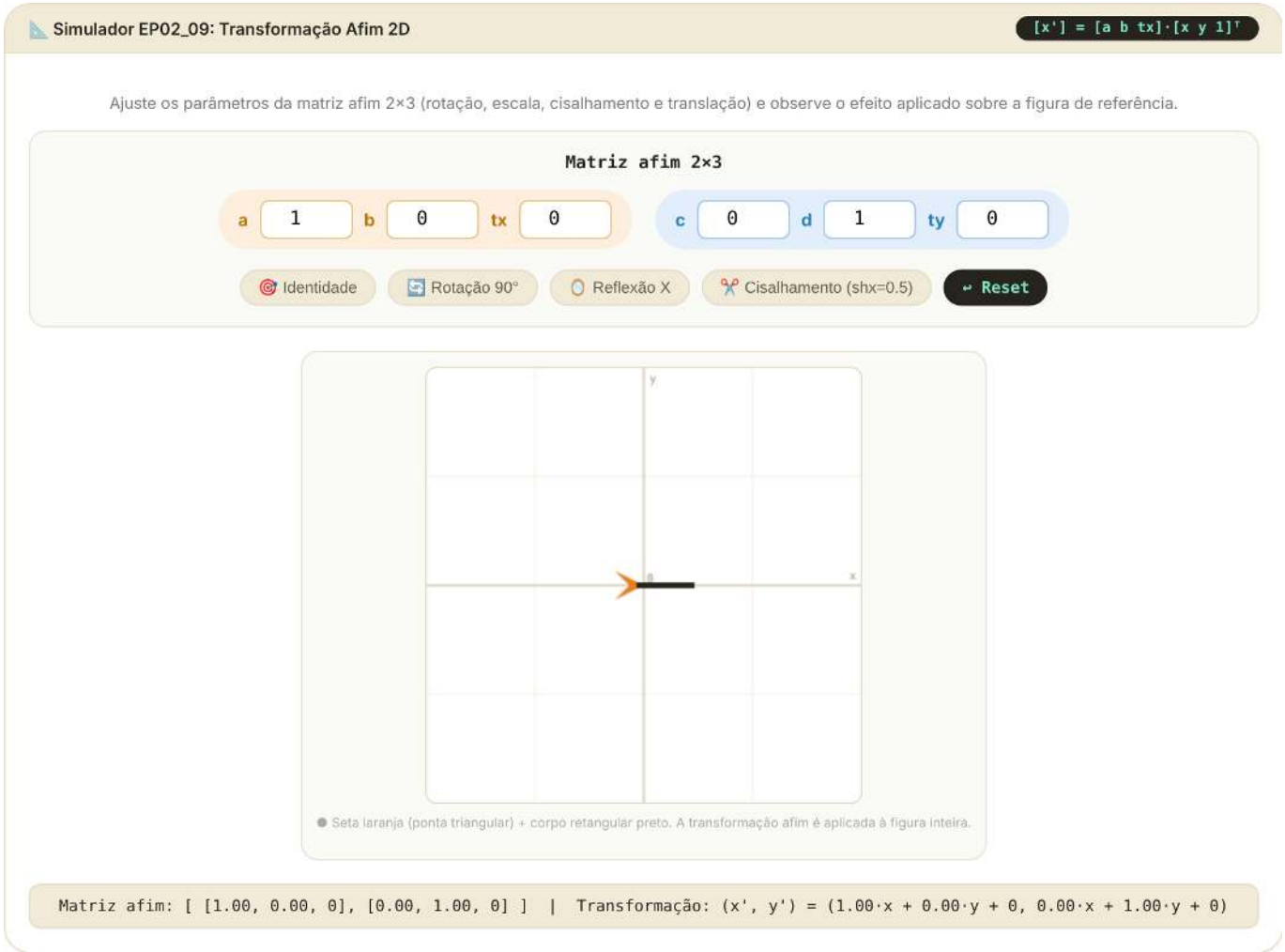
```
TestSuite("EP02_09.cpp").run()
```

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

2.12.10 EP02_10 🎯 Correção de Perspectiva (Homografia)

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

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz original.
- Leia **quatro pares de coordenadas** (x, y) representando os cantos do quadrilátero de origem (objeto distorcido).
- Leia **quatro pares de coordenadas** (x, y) representando os cantos do quadrilátero de destino (onde o objeto deve ser mapeado).



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap02/sim-ep0209-afim.html

Figura 2.20: Simulador EP02_09: Transformação Afim 2D (Matriz 2×3)

- Leia os valores da matriz original.
- Calcule a matriz de homografia 3×3 e aplique a transformação.
- Imprima a matriz resultante com as dimensões de saída especificadas.
- Ver na Figura 2.21 uma simulação deste EP.

📌 **Importante**:

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

```
# Dimensões de saída: bounding box dos pontos destino + 1
w = int(max(pts2[:, 0])) + 1; h = int(max(pts2[:, 1])) + 1
# M = cv2.getPerspectiveTransform(pts1, pts2)
# dst = cv2.warpPerspective(img, M, (w, h))
# ou
dst = mm.perspective_transform(img, pts1, pts2, size=(w, h))
```

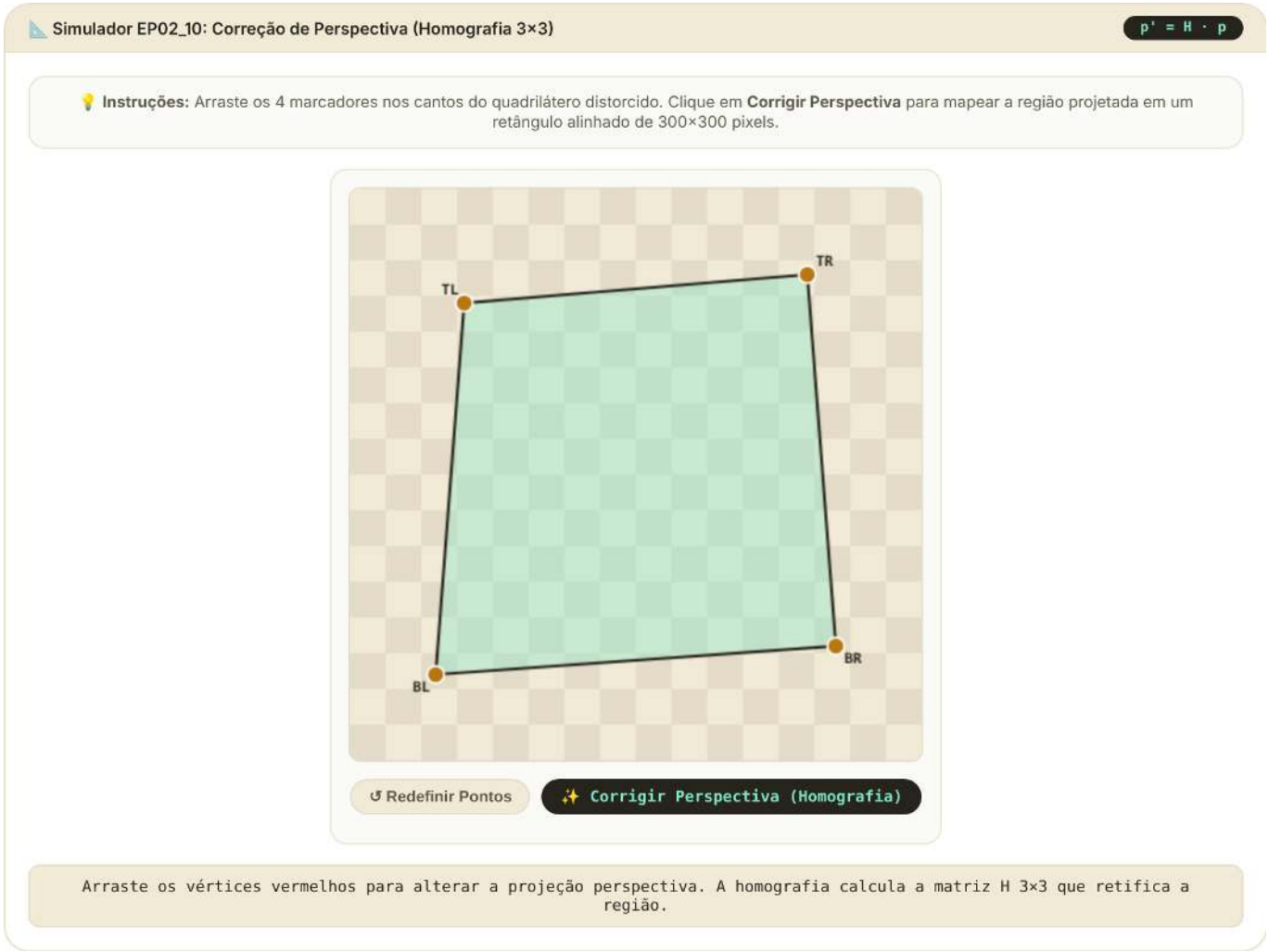
2.12.10.1 🧠 Deformação não-afim

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

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

2.12.10.2 📌 Exemplos

Entrada	Saída	Observação
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	As 4 primeiras linhas após as dimensões são os pontos de origem; as 4 seguintes são os destinos. Com pontos idênticos, a transformação de perspectiva é a identidade e a imagem é preservada.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap02/sim-ep0210-perspectiva.html

Figura 2.21: Simulador EP02_10: Correção de Perspectiva (Transformação de Homografia 3×3)

```
%%writefile EP02_10.cpp
// sua solução
```

```
Overwriting EP02_10.cpp
```

```
TestSuite("EP02_10.cpp").run()
```

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

2.12.11 EP02_11 🏆 Correção de Perspectiva (Homografia) em Imagem Real

Nesta atividade, o objetivo é aplicar a transformação de perspectiva (homografia) para "retificar" um objeto inclinado em uma fotografia real. Você lidará com a imagem de um jornal, onde a grade de um jogo de Sudoku está distorcida devido ao ângulo em que a foto foi capturada.

Seu programa deve ler os parâmetros de entrada do terminal, carregar a imagem, calcular a matriz de homografia 3×3, aplicar a transformação geométrica e exibir um indicador global de validação.

- Leia dois inteiros **L** e **C**, representando as dimensões de linhas e colunas (altura e largura) que a imagem retificada de saída deve ter.
- Leia **quatro pares de coordenadas** (x, y) via terminal, representando os quatro cantos do quadrilátero de origem (o Sudoku distorcido na imagem original).
- Calcule automaticamente os **quatro pares de coordenadas de destino** utilizando as dimensões L e C fornecidas, mapeando os cantos para as extremidades da nova imagem: $(0, 0)$, $(C-1, 0)$, $(0, L-1)$ e $(C-1, L-1)$.
- Carregue a imagem local `sudoku.png` e converta-a para tons de cinza (grayscale).
- Calcule a matriz de homografia e aplique a transformação espacial na imagem.
- **Saída:** Calcule e imprima a **soma de todos os pixels** da imagem resultante.

📌 **Importante**:

- **Arquivo de entrada:** A imagem `sudoku.png` deve estar na mesma pasta do script. O programa deve lê-la diretamente do disco (ex: usando `mm::read("sudoku.png")` ou `cv2.imread`).
- **Ordem dos Pontos:** Garanta que a leitura dos 4 pontos de origem e a geração dos 4 pontos de destino sigam rigorosamente a mesma ordem dos cantos: Superior-Esquerdo (TL), Superior-Direito (TR), Inferior-Esquerdo (BL) e Inferior-Direito (BR).
- **Dimensões no OpenCV:** Lembre-se que funções como `cv2.warpPerspective` esperam o tamanho da imagem de saída no formato `(largura, altura)`, o que equivale a `(C, L)`.
- **Interpolação:** Para garantir a consistência matemática da soma dos pixels com o corretor automático, utilize a interpolação bilinear padrão (`flags=cv2.INTER_LINEAR`).
- **Créditos:** A imagem utilizada é **"Sudoku en periódico"** de Héctor Rodríguez, sob licença **CC BY 2.0**.

2.12.11.1 🧠 Contexto do Problema

A homografia possui 8 graus de liberdade, exigindo no mínimo 4 correspondências de pontos para ser calculada. Ao contrário de transformações afins, ela mapeia qualquer quadrilátero em outro quadrilátero, permitindo que linhas que convergem para pontos de fuga voltem a ser paralelas:

Operação	Característica	Aplicação Típica
Homografia	Projeção entre planos	Retificação de documentos, escaneamento de placas e QR Codes.
Mapeamento Inverso	Varredura do destino para a origem	Evita "buracos" ou pixels vazios na imagem final retificada.

Operação	Característica	Aplicação Típica
Warping	Reamostragem espacial	Correção de distorção de lentes e montagem de panoramas (*stitching*).

2.12.11.2 📌 Exemplos

Entrada	Saída	Observação
500 500 100 120 420 95 80 440 450 460	32982820	As duas primeiras entradas são as dimensões de saída (L e C). As 4 linhas seguintes são as coordenadas (x, y) dos cantos do Sudoku na imagem original + PAD. A saída é a soma total dos pixels da imagem retificada.
200 200 100 120 420 95 80 440 450 460	5277150	Mesmos pontos de origem do exemplo anterior, mas gerando uma imagem de saída menor (200×200). A soma dos pixels reduz proporcionalmente devido à escala.

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

A Figura 2.22 mostra a leitura da imagem original seguida da conversão para tons de cinza e redimensionamento para uma matriz de 500×500 pixels, preparando os dados para a etapa seguinte.

A correção de perspectiva, aplicada na Figura 2.23 via matriz de homografia, elimina as deformações causadas pelo ângulo da câmera e produz uma visão frontal e regular da grade do Sudoku.

```
%%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 <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)")
```

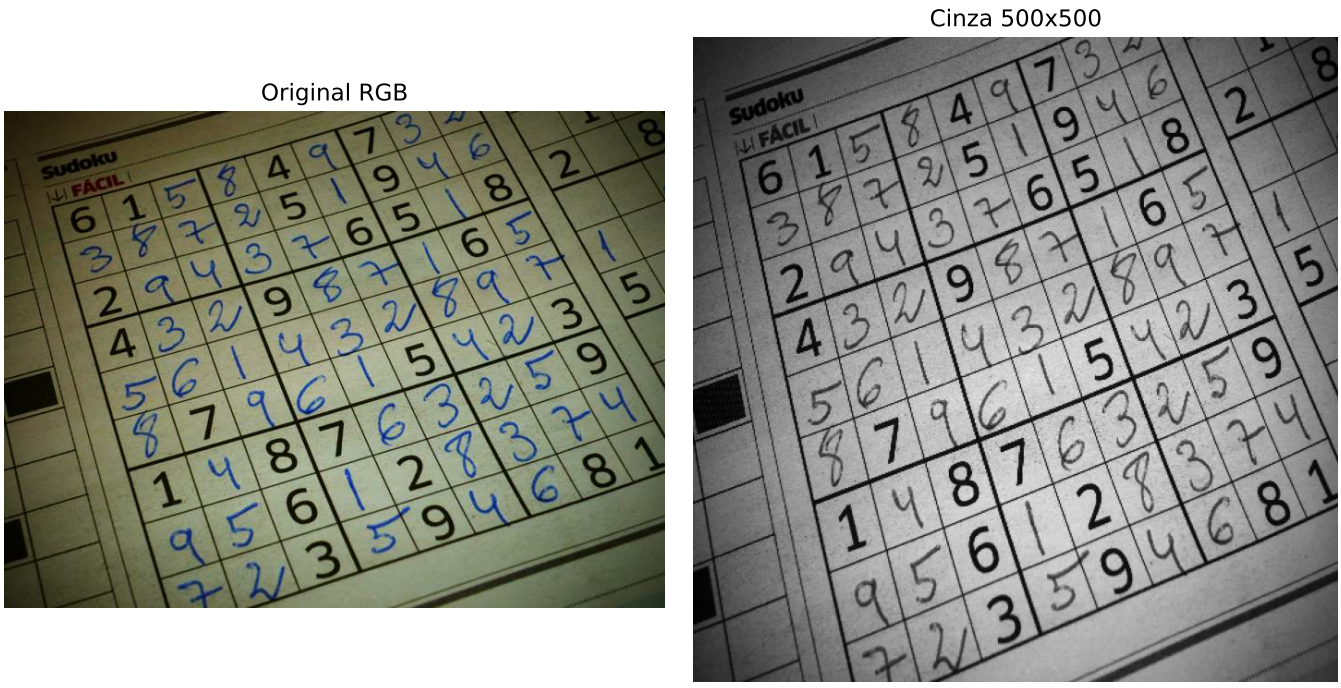


Figura 2.22: Aquisição da imagem de um Sudoku à esquerda. À direita, conversão para tons de cinza e redimensionamento. Crédito: Héctor Rodríguez de Guardamar, Espanha (CC BY 2.0).

```
%%writefile tmp/fig_02_sudoku2.cpp
#define MM_OUT "tmp/fig_02_sudoku2.png"
// | label: fig-02-sudoku2
// | fig-cap: "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)."
// | echo: true
// | output: true

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

int main() {
    // 1. Imagem gravada pela célula anterior (sudoku.png, 500×500, cinza)
    mm::Image img = mm::read("sudoku.png");

    // 2. Padding para não cortar os vértices da grade (mm::pad preenche com 0)
    int PAD = 60;
    mm::Image img_pad = mm::pad(img, PAD);

    // 3. Cantos da grade na imagem expandida - TL, TR, BL, BR
    std::vector<std::array<double, 2>> pts1 = {{100, 160}, {390, 45}, {200, 580}, {570, 420}};

    // 4. Cantos de destino: vista frontal SIZE×SIZE
    int SIZE = 500;
    std::vector<std::array<double, 2>> pts2 = {{0, 0}, {SIZE, 0}, {0, SIZE}, {SIZE, SIZE}};

    // 5. Homografia pts1 -> pts2 e retificação
    mm::Image img_rect = mm::perspective_transform(img_pad, pts1, pts2, SIZE, SIZE);

    // 6. Exibição
```

```
    mm::show(
        std::vector<mm::Image>{img_pad, img_rect},
        MM_OUT,
        std::vector<std::string>{"Original (com padding)", "Vista frontal retificada"},
        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"
```

```
tmp/fig_02_sudoku2.cpp: In function ‘int main()’:
tmp/fig_02_sudoku2.cpp:26:57: warning: narrowing conversion of ‘SIZE’ from ‘int’ to ‘double’
[-Wnarrowing]
   26 |     std::vector<std::array<double, 2>> pts2 = {{0, 0}, {SIZE, 0}, {0, SIZE}, {SIZE,
   SIZE}};
      |                                                         ^~~~
tmp/fig_02_sudoku2.cpp:26:71: warning: narrowing conversion of ‘SIZE’ from ‘int’ to ‘double’
[-Wnarrowing]
   26 | ector<std::array<double, 2>> pts2 = {{0, 0}, {SIZE, 0}, {0, SIZE}, {SIZE, SIZE}};
      |                                                               ^~~~

tmp/fig_02_sudoku2.cpp:26:79: warning: narrowing conversion of ‘SIZE’ from ‘int’ to ‘double’
[-Wnarrowing]
   26 | d::array<double, 2>> pts2 = {{0, 0}, {SIZE, 0}, {0, SIZE}, {SIZE, SIZE}};
      |                                                               ^~~~

tmp/fig_02_sudoku2.cpp:26:85: warning: narrowing conversion of ‘SIZE’ from ‘int’ to ‘double’
[-Wnarrowing]
   26 | :array<double, 2>> pts2 = {{0, 0}, {SIZE, 0}, {0, SIZE}, {SIZE, SIZE}};
      |                                                                 ^~~~

[1] Original (com padding)
[2] Vista frontal retificada
```

```
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)")
```

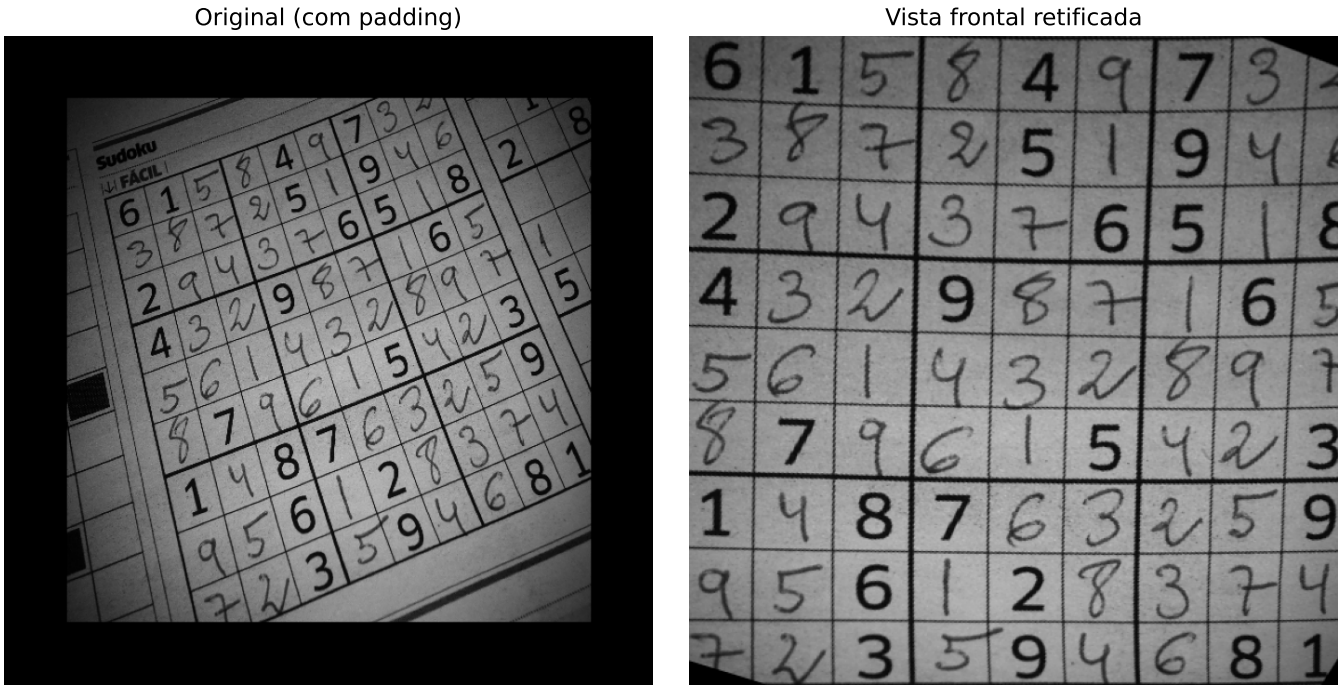


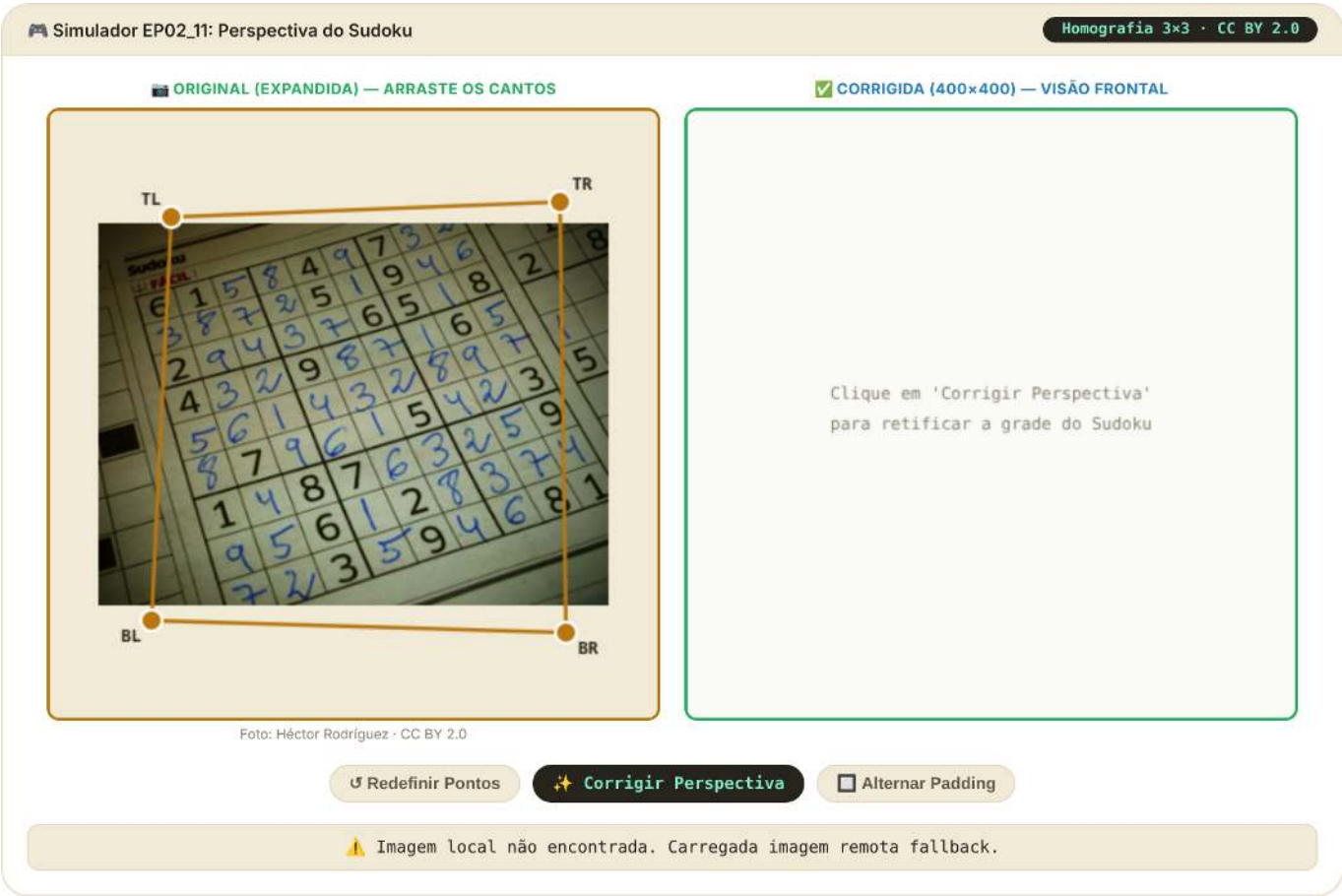
Figura 2.23: Correção de perspectiva por homografia 3×3: imagem com *padding* e a vista frontal retificada, via *mm::perspective_transform* (solve 8×8 dos 4 pontos + mapeamento inverso projetivo).

```
%%writefile EP02_11.cpp
// sua solução
```

```
Overwriting EP02_11.cpp
```

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

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap02/sim-ep0211-sudoku.html

Figura 2.24: Simulador EP02_11: Correção de Perspectiva do Sudoku (Homografia 3×3 com Reamostragem Bilinear)

Capítulo 3

Operações Espaciais: Intensidade, Histograma e Filtragem

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

3.1 Objetivos

Ao final deste capítulo, você será capaz de:

- **Manipular intensidade e pixels:** Executar operações aritméticas saturadas (`mm::addm`, `mm::subm`) e lógicas bit a bit (`mm::band`, `mm::bor`, `mm::bnot`) para combinação e seleção de regiões de interesse (ROI), e aplicar *alpha blending* (`mm::blend`) para fusão ponderada de imagens;
- **Processar histogramas:** Interpretar o histograma como diagnóstico tonal e aplicar equalização global via CDF (`mm::equalize`); na trilha Python, também a equalização adaptativa (CLAHE) e a especificação de histograma para transferência de perfil tonal entre imagens;
- **Compreender fundamentos espaciais:** Entender vizinhança, *padding* de borda (`mm::pad`) e a diferença entre correlação cruzada (`mm::conv`) e convolução — incluindo por que *kernels* assimétricos como o de Sobel produzem resultados distintos nas duas operações;
- **Aplicar filtragem de suavização:** Usar o filtro de média (`mm::blur`, ou `mm::conv` com *kernel* uniforme) e o filtro Gaussiano (`mm::gaussian`) para redução de ruído, compreendendo a vantagem da ponderação radial e da separabilidade Gaussiana;
- **Aplicar filtragem de realce:** Usar o Laplaciano w_4 e w_8 (`mm::laplacian`) para realce isotrópico de bordas, o operador de Sobel (`mm::sobel`) para a magnitude do gradiente — e, na trilha Python, a decomposição direcional G_x, G_y e ângulo —, e o *Unsharp Masking* (`mm::usm`) para amplificação de alta frequência controlada pelo parâmetro k;
- **Utilizar filtros de ordem:** Aplicar o filtro da mediana (`mm::median`) para remoção de ruído sal e pimenta, compreendendo por que sua natureza não linear e a robustez a *outliers* o tornam superior aos filtros lineares nesse cenário;
- **Resolver problemas práticos:** Encadear técnicas em *pipelines* de pré-processamento (equalização → Gaussiano → Canny; com CLAHE no lugar da equalização na trilha Python) e usar as funções da `morph` (`mm::conv`, `mm::histImg`, `mm::equalize`, `mm::drawImgKernel`) para análise e visualização didática de cada etapa.

3.2 Operações em Nível de Intensidade

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

Formalmente, uma transformação de ponto pode ser descrita como:

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

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

3.2.1 Preparando o Ambiente Prático

O bloco a seguir carrega a biblioteca `morph` do repositório (o módulo `morph.py` e, na trilha C++, também a `morph.hpp` usada no `#include` das células compiladas).

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # artefatos de build da trilha C++ (.cpp, binário,
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")

# O kernel é Python mesmo na trilha C++: `mm` (morph.py) é usado pelos
# simuladores, pela exibição das figuras que o binário C++ gera e pelo
# estado mm::Image entre células. cpp=True baixa também a trilha compilada
# (morph.hpp + stb_image*.h), usada no #include das células %%writefile *.cpp.
import config
config.setup(cpp=True)
from morph import mm
import numpy as np
```

```
Ambiente pronto. Morph: 1.1.9 | OpenCV: 5.0.0
```

Como objeto de estudo ao longo deste capítulo, utilizaremos as imagens de vida selvagem apresentadas nas Figura 3.1 e Figura 3.3. A partir delas, exploraremos operações espaciais sobre intensidade, histogramas e filtragem, analisando seus efeitos no realce, na suavização, na redução de ruído e na detecção de bordas, de modo a compreender os fundamentos matemáticos e computacionais do PDI.

```
%%writefile tmp/fig_03_mandrill.cpp
#define MM_OUT "tmp/fig_03_mandrill.png"
// Compile: g++ -std=c++17 -o programa programa.cpp -I. -lstdc++fs

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

int main() {
    // | label: fig-03-mandrill
    // | fig-cap: "*Mandrill* (*Mandrillus sphinx*) fotografado em ambiente natural na África
    do Sul. Crédito: Carlos Guilherme Rodrigues (CC BY-SA 3.0)."
    // | echo: true

    std::string base    = "https://upload.wikimedia.org/wikipedia/commons";
    std::string arquivo = "Carlos_Guilherme_Rodrigues_%2876515283%29.jpeg";
    std::string url     = base + "/9/9b/" + arquivo;
    std::string caminho = "imagens/mandrill-exif.jpg";

    if (!std::filesystem::exists(caminho)) {
        std::filesystem::create_directories("imagens");
        mm::write(mm::read(url), caminho);
```

```
    }

    mm::Image img_color = mm::read(caminho);
    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)")
```

Figura 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 Operações Aritméticas

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

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

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

⚠ Saturação e *overflow*

Operações aritméticas em `uint8` sofrem *overflow* silencioso: $200 + 100 = 44$ (não 300). `mm::addm` e `mm::subm` fazem a **saturação automática**, fixando o resultado em $[0, 255]$. O *blending* usa pesos fracionários: `mm::blend` opera internamente em ponto flutuante e só então arredonda e satura para `uint8`.

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

```
%%writefile tmp/fig_03_aritmetica.cpp
#define MM_OUT "tmp/fig_03_aritmetica.png"
#include <iostream>
#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_8.png");
// [pdi:state-io:end]

    // Variáveis fornecidas automaticamente:
    // img_gray (mm::Image)

    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)")
```

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

3.2.3 Mistura Ponderada (*Alpha Blending*)

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

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

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

Para que a combinação produza um resultado coerente, é necessário alinhar previamente as regiões de interesse. Na Figura 3.4, recorta-se o rosto do leopardo com `mm::crop(img_leop_gray, 250, H-300, 100, W-200)` e a região facial do mandril com `mm::crop(img_gray, 100, 400, 380, 530)`, de modo que olhos e estrutura facial fiquem aproximadamente alinhados. O recorte do leopardo é então redimensionado (`mm::resize`) para as dimensões do mandril antes da mistura.

`mm::blend` faz a operação em ponto flutuante — evitando *overflow* nas contas com pesos fracionários — e só então arredonda e satura o resultado para `uint8`.

```
%%writefile tmp/fig_03_leopardo.cpp
#define MM_OUT "tmp/fig_03_leopardo.png"
#include "morph.hpp"
#include <iostream>
#include <filesystem>

int main() {
    //| 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

    const std::string base    = "https://upload.wikimedia.org/wikipedia/commons";
    const std::string arquivo = "Leopard_%28Panthera_pardus%29_portrait.jpg";
    const std::string url     = base + "/9/92/" + arquivo;

    mm::Image img_leop      = mm::read(url);
    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)")
```

Figura 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 <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");
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(
        {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,
        {" =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)")
```

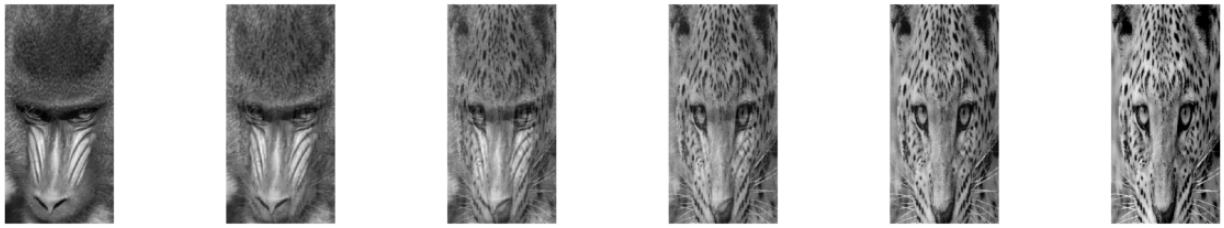


Figura 3.4: *Alpha blending* entre recortes alinhados de mandrill e do leopardo (Figura 3.3) para diferentes valores de . Em =1 vê-se apenas mandrill; em =0, apenas o leopardo; valores intermediários fundem os olhares das duas imagens proporcionalmente.

3.2.4 Operações Lógicas e Máscaras Bit a Bit

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

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

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

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

```
%%writefile tmp/fig_03_logica.cpp
#define MM_OUT "tmp/fig_03_logica.png"
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#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]

    // img_gray já está inicializado

    int h = img_gray.h;
    int w = img_gray.w;

    // Máscara circular preenchida, centrada na imagem (mm.circle desenha o
    // disco; a versão didática, teste de raio pixel a pixel, é 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);

    // Operações via morph
    mm::Image img_not = mm::bnot(img_gray);            // NOT: negativo fotográfico
    mm::Image img_and = mm::band(img_gray, mask_circ); // preserva apenas a ROI circular
    mm::Image img_or  = mm::bor(img_gray, mask_circ);  // ilumina a região da máscara

    mm::show(
```

```
        std::vector<mm::Image>{img_gray, img_and, img_or, img_not},
        MM_OUT,
        std::vector<std::string>{"Original", "AND (ROI circular)", "OR (ilumina ROI)", "NOT
(negativo)"},
        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 (ROI circular)
[3] OR (ilumina ROI)
[4] NOT (negativo)
```

```
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)")
```

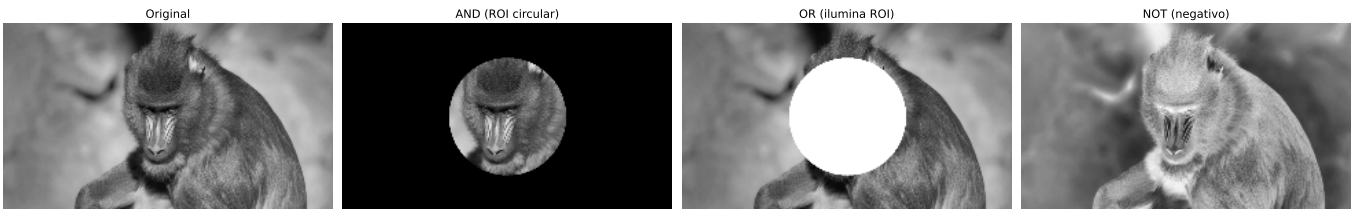


Figura 3.5: Operações lógicas bit a bit com máscara circular: AND (isolamento da ROI), OR (iluminação da ROI) e NOT (negativo).

3.3 Histograma de Imagens

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

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

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

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

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

i Interpretação do Histograma

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

A Figura 3.6 apresenta o histograma da imagem do mandrill, bem como versões escurecida (`mm::subm`) e clareada (`mm::addm`). Observa-se o deslocamento da distribuição de intensidades para a esquerda e para a direita, respectivamente. Note que o intervalo representado no eixo x não corresponde necessariamente a toda a faixa de 0 a 255.

```
%%writefile tmp/fig_03_histograma.cpp
#define MM_OUT "tmp/fig_03_histograma.png"
#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 já está inicializado (inserido automaticamente)

    // Versão escurecida (-80) e clareada (+80) com saturação
    mm::Image img_dark = mm::subm(img_gray, 80);
    mm::Image img_high = mm::addm(img_gray, 80);

    // Exibição das imagens e seus histogramas
    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)")
```

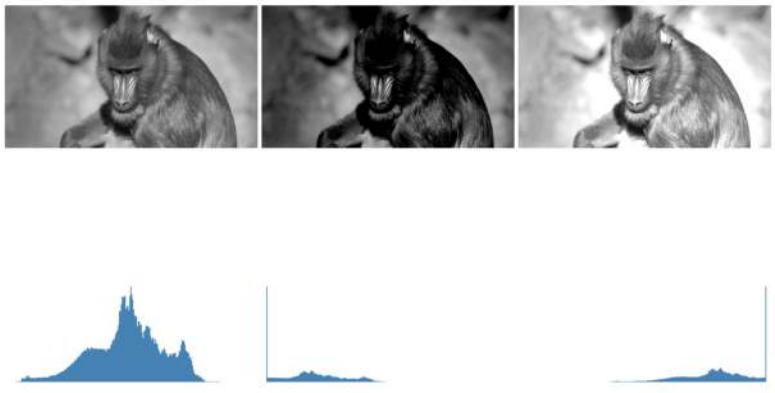


Figura 3.6: Histogramas da imagem original, de uma versão escurecida (-80) e de uma clareada ($+80$). A subtração/adição satura em 0 e 255.

3.3.1 Equalização de Histograma

A **equalização de histograma** redistribui as intensidades para que o histograma resultante seja o mais uniforme possível. O mapeamento é dado pela **função de distribuição acumulada (CDF)**:

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p(r_j) = \frac{L-1}{MN}\sum_{j=0}^{k} n_j \tag{3.7}$$

A transformação é monotônica: níveis frequentes recebem intervalos maiores no domínio de saída (maior separação $\rightarrow$ mais contraste), enquanto níveis raros são comprimidos.

O algoritmo completo, em cinco etapas, é apresentado na Tabela 3.1.

Tabela 3.1: Algoritmo de equalização de histograma.

Etapa	Operação	Fórmula
1	**Histograma**	$h[k] \leftarrow$ número de pixels com intensidade k, $k = 0 \dots L-1$
2	**Probabilidade**	$p[k] \leftarrow h[k]/MN$
3	**CDF**	$\text{cdf}[k] \leftarrow \sum_{j=0}^{k} p[j]$ (soma acumulada)
4	***Look-Up Table* (mapeamento)**	$\text{lut}[k] \leftarrow \text{round}(\text{cdf}[k] \times (L-1))$
5	**Aplicação**	$g[i,j] \leftarrow \text{lut}[f[i,j]]$ (para todo pixel)

Note na Figura 3.7 que a equalização **redistribui** os tons existentes para posições mais espaçadas na faixa $[0, L-1]$, mas não cria novos tons — a imagem equalizada continua com exatamente 3 tons distintos, agora em $\{1, 5, 7\}$ em vez de $\{2, 3, 4\}$.

```
%%writefile tmp/fig_03_equalizacao_didatica.cpp
#define MM_OUT "tmp/fig_03_equalizacao_didatica.png"
//| 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

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

int main() {
    mm::Image img5(5, 5);
    // preenche a imagem 5x5 com os valores especificados
    unsigned char 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) << "\n";
    std::cout << "Imagem equalizada 5x5:\n";
    std::cout << mm::drawImg(img5_eq) << "\n";

    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)")
```

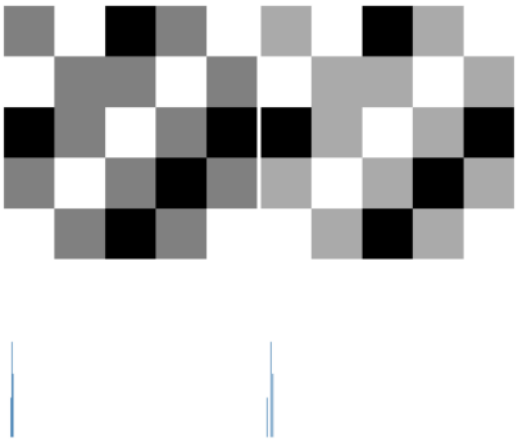


Figura 3.7: Equalização de histograma numa imagem 5×5 de 3 bits (L=8): tons concentrados em {2,3,4} são redistribuídos pela CDF. *mm::equalize(img, 3)* faz o mapeamento.

Limitação: a equalização global pode super-realçar ruídos e produzir contraste excessivo em regiões homogêneas. O **CLAHE** (*Contrast Limited Adaptive Histogram Equalization*) reduz esse problema ao aplicar a equalização em blocos locais (*tiles*) e limitar a altura dos picos do histograma antes da equalização.

A Figura 3.8 compara a imagem original, a equalização global via `mm::equalize` e o CLAHE do OpenCV,

exibindo também os histogramas resultantes. Diferentemente da equalização global, que utiliza uma única transformação baseada na CDF de toda a imagem, o CLAHE adapta o contraste a cada região, sendo particularmente útil em imagens com iluminação não uniforme.

No exemplo, foi utilizado `clipLimit=2.0` e `tileGridSize=(32,32)`. O parâmetro `clipLimit` define o quanto os picos do histograma local podem crescer antes de serem cortados (*clipped*). No OpenCV, esse valor é um fator relativo: o limite real é aproximadamente calculado como `clipLimit × (número de pixels do bloco / número de níveis de cinza)`. Por exemplo, em um bloco com 4096 pixels e uma imagem de 8 bits (256 níveis de cinza), a frequência média por nível é $4096/256 = 16$. Assim, `clipLimit=2.0` permite picos de aproximadamente $2 \times 16 = 32$ ocorrências antes do corte. As ocorrências excedentes não são descartadas: elas são redistribuídas entre os demais níveis de cinza do histograma, reduzindo a concentração excessiva em poucos níveis e evitando uma amplificação exagerada do contraste local. Valores menores limitam mais o contraste e reduzem a amplificação de ruído, enquanto valores maiores permitem um realce mais intenso, mas podem introduzir artefatos.

- `clipLimit=1.0`: realce suave e conservador;
- `clipLimit=2.0`: bom equilíbrio entre contraste e naturalidade;
- `clipLimit=4.0`: maior destaque para detalhes locais;
- `clipLimit=8.0`: contraste agressivo, com possível amplificação de ruído.

Assim, o CLAHE costuma produzir resultados mais naturais do que a equalização global, especialmente em imagens com sombras, reflexos ou iluminação desigual.

```
%%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 <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_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)")
```

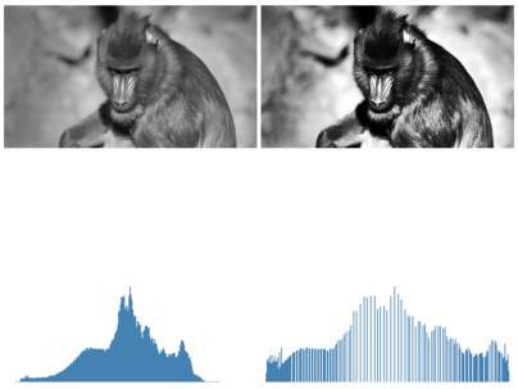


Figura 3.8: Equalização de histograma global (mm::equalize, via CDF) e os histogramas antes/depois. CLAHE (adaptativa) fica só na trilha Python — não tem equivalente em morph.hpp.

3.3.2 Especificação de Histograma

Enquanto a equalização impõe uma distribuição uniforme, a **especificação de histograma** (*histogram matching*) permite que o histograma da imagem de saída siga uma distribuição **arbitrária** — por exemplo, o histograma de outra imagem de referência.

O procedimento envolve três etapas:

1. Calcular a CDF da imagem de entrada: $P_r(r_k)$.
2. Calcular a CDF da imagem de referência: $P_z(z_k)$.
3. Para cada nível r_k, encontrar o nível z que minimiza $|P_z(z) - P_r(r_k)|$.

$$T(r_k) = \arg\min_z |P_z(z) - P_r(r_k)| \tag{3.8}$$

Na Figura 3.9, transferimos o perfil tonal do leopardo (Figura 3.3) para a imagem do mandrill — uma aplicação direta do conceito visto no *blending*: em vez de fundir pixels, aqui fundimos distribuições tonais.

```
%%writefile tmp/fig_03_especificacao.cpp
#define MM_OUT "tmp/fig_03_especificacao.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 = 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 e img_leop_gray são fornecidos automaticamente

    // Equalização global do mandril
    mm::Image img_eq = mm::equalize(img_gray);

    // Exibição das imagens: original, referência e equalizada
    mm::show(std::vector<mm::Image>{img_gray, img_leop_gray, img_eq},
             MM_OUT,
             std::vector<std::string>{"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)")
```

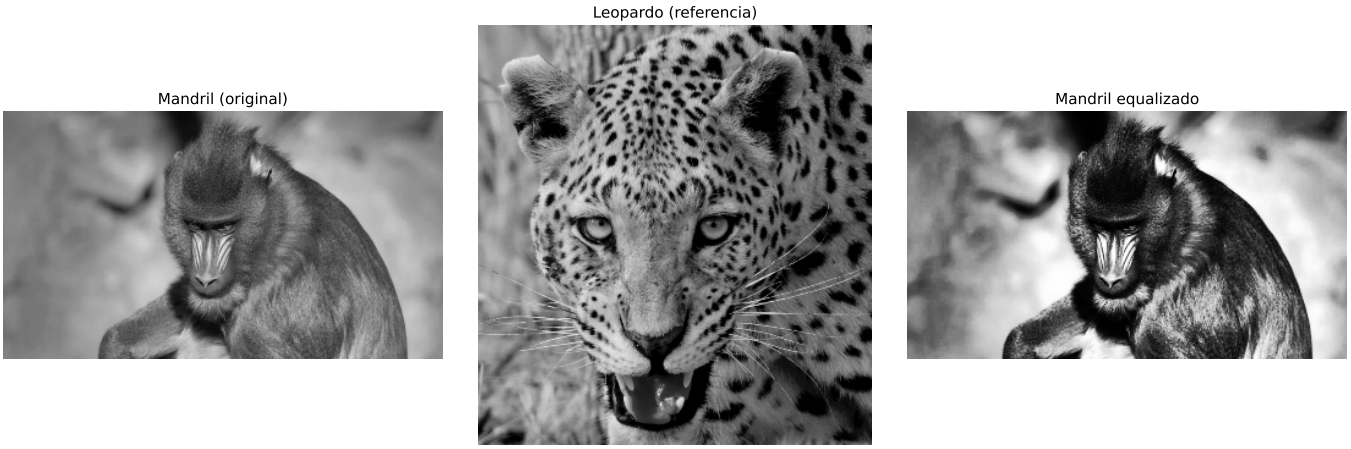


Figura 3.9: Especificação de histograma (mapear o mandril para o perfil tonal do leopardo) precisa da CDF inversa da referência — fica só na trilha Python. Aqui, a equalização global do mandril, como comparação.

3.4 Fundamentos Espaciais: Vizinhança, Convolução e *Kernels*

As operações de filtragem espacial não atuam em um único pixel isolado, mas em uma **vizinhança** ao seu redor. Para isso, utiliza-se uma pequena matriz de coeficientes denominada **kernel** (ou máscara), que percorre toda a imagem por meio de uma **janela deslizante** (*sliding window*).

As janelas mais comuns são 3×3, 5×5 e 7×7. Em uma janela 3×3, por exemplo, o pixel central é processado juntamente com seus oito vizinhos imediatos. Em cada posição da janela, os valores dos pixels são combinados com os coeficientes do *kernel*, produzindo um novo valor para o pixel central.

3.4.1 Vizinhança

Considere uma janela 3×3 centrada no pixel (x, y):

$$\begin{bmatrix} (x-1, y-1) & (x, y-1) & (x+1, y-1) \\ (x-1, y) & (x, y) & (x+1, y) \\ (x-1, y+1) & (x, y+1) & (x+1, y+1) \end{bmatrix} \tag{3.9}$$

De forma geral, uma janela de tamanho $(2a+1) \times (2b+1)$ abrange todos os pixels situados até a posições na horizontal e até b posições na vertical em relação ao pixel central. Assim, uma janela 3×3 corresponde a $a = b = 1$, uma janela 5×5 a $a = b = 2$, e assim por diante.

Matematicamente, a vizinhança é definida por

$$\mathcal{V}(x, y) = \{(x+s,\, y+t) : -a \leq s \leq a,\, -b \leq t \leq b\} \tag{3.10}$$

3.4.2 Tratamento de Bordas

Pixels próximos às bordas possuem parte de sua vizinhança fora da imagem. Para aplicar filtros nessas regiões, é preciso definir como os valores externos serão obtidos. As três estratégias mais comuns (com a constante equivalente do OpenCV entre parênteses) são:

- **Zero-padding** (`BORDER_CONSTANT`): completa a região externa com zeros.
- **Replicação** (`BORDER_REPLICATE`): repete o valor do pixel da borda.
- **Reflexão** (`BORDER_REFLECT_101`): espelha os pixels vizinhos, sem repetir o da borda.

`mm::conv` usa a reflexão por padrão, pois preserva melhor a continuidade dos níveis de cinza e reduz artefatos no tratamento das bordas.

O exemplo a seguir compara as três estratégias com `mm::pad` numa matriz 3×3. Observe como cada uma preenche os pixels externos necessários para aplicar um filtro 3×3 também nos cantos.

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

int main() {
    // Criando a imagem 3x3 diretamente
    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:" << std::endl;
    std::cout << mm::drawImg(img) << std::endl;

    for (int k : {3, 5}) {
        int b = k / 2;
        std::cout << "=== Kernel " << k << "x" << k << " (padding b=" << b << ") ===" <<
std::endl;
        for (const std::string& nome : bordas) {
            std::cout << nome << std::endl;

            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)) << std::endl;
        }
    }

    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
```

Observe que o resultado de um filtro pode variar significativamente conforme o tratamento adotado para as bordas da imagem.

Na `morph.hpp`, as funções de filtragem (`mm::conv`, `mm::blur`, `mm::gaussian`, `mm::laplacian`, `mm::usm`) aplicam *padding* por **reflexão** (`mm::Border::REFLECT101`) por padrão — o mesmo comportamento do `cv2.filter2D`. A variante didática `mm::conv0` usa `mm::Border::KEEP`: os pixels da borda mantêm o valor original, sem o filtro. Já `mm::sobel` e `mm::prewitt` deixam a borda em zero (calculam apenas o interior).

3.4.3 Correlação vs. Convolução

Existem dois mecanismos matematicamente relacionados.

Correlação cruzada (*cross-correlation*) — o *kernel* é aplicado diretamente:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)\, f(x+s,\, y+t) \tag{3.11}$$

Convolução bidimensional — o *kernel* é rotacionado 180° antes da aplicação:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)\, f(x-s,\, y-t) \tag{3.12}$$

Para *kernels* simétricos (Gaussiano, Laplaciano, média) as duas operações produzem resultados idênticos. Para *kernels* assimétricos (Sobel, Prewitt) a diferença é significativa, como mostram os exemplos a seguir.

3.4.3.1 Correlação (`mm::conv`)

```
%%writefile tmp/mm_out_2.cpp
// Compile: g++ -std=c++17 -o programa programa.cpp -I. -lmorph

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

int main() {
    // imagem 4x4 (uint8) e kernel assimétrico 3x3
    mm::Image img(4, 4, 1);
    unsigned char dados[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) = dados[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:" << std::endl;
    std::cout << mm::drawImg(img) << std::endl;
    std::cout << "Kernel:" << std::endl;
    // Como a função drawImg espera mm::Image, precisamos converter o kernel
    mm::Image kernel_img(3, 3, 1);
    for (int y = 0; y < 3; ++y)
        for (int x = 0; x < 3; ++x)
            kernel_img.at(y, x, 0) = (unsigned char)w.at(y, x);
    std::cout << mm::drawImg(kernel_img) << std::endl;
    std::cout << "Resultado da correlação:" << std::endl;
    std::cout << mm::drawImg(corr) << 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
```

```
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` realiza correlação, isto é, aplica o *kernel* exatamente na orientação fornecida.

3.4.3.2 Convolução

```
%%writefile tmp/mm_out_3.cpp
#include "morph.hpp"
#include <iostream>

int main() {
    // repetidos aqui para a célula ser independente
    mm::Image img(4, 4);
    // Preenchendo a imagem com valores 1..16
    int val = 1;
    for (int y = 0; y < 4; y++) {
        for (int x = 0; x < 4; x++) {
            img.at(y, x) = val++;
        }
    }

    mm::Kernel w{{0, 1, 2},
                 {0, 0, 0},
                 {0, 0, 0}};

    // kernel rotacionado 180° (equivale a np.rot90(w, 2))
    mm::Kernel w_conv{{0, 0, 0},
                      {0, 0, 0},
                      {2, 1, 0}};

    mm::Image conv = mm::conv(img, w_conv, mm::Border::CONSTANT);

    std::cout << "Imagem original:" << "\n";
    std::cout << mm::drawImg(img) << "\n";
    std::cout << "Kernel original:" << "\n";
    std::cout << mm::drawImg(w) << "\n";
    std::cout << "Kernel rotacionado 180°:" << "\n";
    std::cout << mm::drawImg(w_conv) << "\n";
    std::cout << "Resultado da convolução:" << "\n";
    std::cout << mm::drawImg(conv) << "\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
```

A convolução utiliza o *kernel* rotacionado em 180°. Para reproduzir a definição matemática de convolução, rotaciona-se o *kernel* (aqui, `[[0,1,2],[0,0,0],[0,0,0]]` → `[[0,0,0],[0,0,0],[2,1,0]]`) antes de aplicar `mm::conv`.

3.4.4 O Papel do *Kernel*

Os coeficientes do *kernel* determinam completamente o efeito produzido pelo filtro, conforme resumido na Tabela 3.2.

Tabela 3.2: Interpretação típica dos coeficientes do *kernel*.

Característica	Efeito típico
Coeficientes positivos com soma 1	Suavização (*passa-baixa*)
Soma igual a 0, com valores positivos e negativos	Detecção de bordas (*passa-alta*)
Coeficiente central positivo dominante e vizinhos negativos	Realce de nitidez
Coeficientes assimétricos	Gradiente direcional

Exemplos:

Suavização:

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

Detecção de bordas:

$$\begin{bmatrix}-1 & -1 & -1\\ -1 & 8 & -1\\ -1 & -1 & -1\end{bmatrix}$$

Realce de nitidez:

$$\begin{bmatrix}0 & -1 & 0\\ -1 & 5 & -1\\ 0 & -1 & 0\end{bmatrix}$$

Gradiente direcional (Sobel):

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

A Figura 3.10 demonstra o mecanismo passo a passo: para cada posição da janela, multiplica-se cada coeficiente do *kernel* pelo pixel correspondente da vizinhança e somam-se os produtos obtidos. O resultado é exatamente o valor definido pela Equação 3.11 para aquela posição da imagem. Embora as imagens produzidas por `mm::conv0` e `cv2.filter2D` (ou `mm::conv`) sejam visualmente muito semelhantes, a implementação baseada em OpenCV é milhares de vezes mais rápida, como mostrado a seguir.

⚠ Desempenho: laços Python vs. operações vetorizadas

A função `mm::conv0` implementa a correlação diretamente em Python por meio de laços aninhados. Embora essa abordagem seja adequada para fins didáticos, ela executa um grande número de operações e torna-se lenta para imagens maiores.
Já `mm::conv` utiliza `cv2.filter2D`, implementado em C++ e otimizado para operações matriciais. No exemplo apresentado, a versão vetorizada foi mais de **3000 vezes mais rápida** que a implementação didática, produzindo um resultado visualmente equivalente.
As diferenças numéricas observadas concentram-se principalmente nas bordas da imagem. Em `mm::conv0`, os pixels da borda permanecem inalterados, enquanto `mm::conv` utiliza uma estratégia de reflexão das bordas (`cv2.BORDER_REFLECT_101`, padrão do `cv2.filter2D`).
Por isso, `mm::conv0` deve ser utilizado para compreender o algoritmo, enquanto `mm::conv` é a opção recomendada para aplicações práticas.

```
%%writefile tmp/fig_03_convolucao_passo.cpp
#define MM_OUT "tmp/fig_03_convolucao_passo.png"
//| 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

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

    mm::Kernel w_mean = mm::Kernel::mean(3);
    mm::Image img_gray = img_leop_gray;

    mm::Image img_conv0 = mm::conv0(img_gray, w_mean);   // laços, bordas preservadas
    mm::Image img_conv = mm::conv(img_gray, w_mean);    // borda refletida

    std::cout << "Correlacao no pixel central [251,251]:" << "\n";
    std::cout << "  original = " << (int)img_gray.at(251, 251) << "\n";
    std::cout << "  conv0    = " << (int)img_conv0.at(251, 251) << "\n";
    std::cout << "  conv     = " << (int)img_conv.at(251, 251) << "\n";

    mm::show({img_gray, img_conv0, img_conv},
             MM_OUT,
             {"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)")
```

Figura 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 <filesystem>

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]

    //| echo: false
    // A partir daqui o "sujeito" dos exemplos de filtragem passa a ser o
    // leopardo (mais textura e bordas que o mandril).
    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 Exemplo Numérico: Correlação Passo a Passo

Para tornar concreto o mecanismo da Equação 3.11, considere o *kernel* de média 3×3 ($a = b = 1$, todos os coeficientes $= 1/9 \approx 0{,}111$) aplicado ao *patch* 5×5 extraído da imagem do leopardo. A Figura 3.11 exibe o *patch* com grade e destaca em amarelo a janela 3×3 centrada no pixel $[1, 1]$:

```
%%writefile tmp/fig_03_patch.cpp
#define MM_OUT "tmp/fig_03_patch.png"
#include "morph.hpp"
#include <iostream>

//| 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

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]

    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) << "\n";

    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)")
```

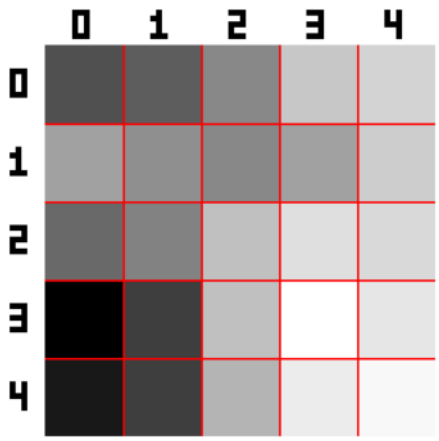


Figura 3.11: *Patch* 5×5 extraído da imagem do leopardo (posição [250:255, 250:255]). A janela amarela destaca a vizinhança 3×3 centrada no pixel [1,1] onde a correlação será calculada.

Para ilustrar o cálculo da correlação, considere o pixel na posição $[1, 1]$ do *patch* 5×5 mostrado na Figura 3.11. Essa posição foi escolhida apenas por conveniência didática, pois possui uma vizinhança 3×3 completa ao seu redor.

Os valores dessa vizinhança correspondem à submatriz superior esquerda do *patch*:

$$\text{vizinhança} = \begin{bmatrix} 91 & 95 & 108 \\ 106 & 107 & 108 \\ 102 & 103 & 107 \end{bmatrix}$$

Aplicando a Equação 3.11 com o kernel de média:

$$g[1,1] = \frac{91+95+108+106+107+108+102+103+107}{9} = \frac{927}{9} = 103$$

O resultado (103) é ligeiramente menor que o valor original do pixel central (107), pois a média incorpora vizinhos de menor intensidade, produzindo o efeito de suavização. Na prática, o algoritmo inicia o processamento em $[0,0]$ e repete esse mesmo cálculo para cada posição da imagem, deslocando a janela até cobrir todo o domínio.

3.5 Filtragem Espacial de Suavização

Os filtros de suavização (*smoothing filters*) atenuam variações bruscas de intensidade, reduzindo ruído e detalhes de alta frequência. São filtros **passa-baixa** — preservam as componentes de baixa frequência (estruturas grandes) e atenuam as de alta frequência (ruído, bordas).

3.5.1 Filtro de Média (*Box Filter*)

O filtro de média utiliza um *kernel* uniforme de tamanho $n \times n$, onde todos os coeficientes valem $1/n^2$:

$$w_{\text{média}} = \frac{1}{n^2}\begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}_{n\times n} \tag{3.13}$$

Cada pixel de saída é a média aritmética dos n^2 pixels de sua vizinhança. Note que a soma dos coeficientes é sempre 1 — o brilho médio da imagem é preservado. *Kernels* maiores produzem suavização mais agressiva, mas borram progressivamente as bordas.

A Figura 3.12 mostra o efeito do filtro de média com *kernels* 3×3, 7×7 e 15×15 sobre um detalhe da imagem do leopardo. Os resultados foram obtidos com `mm::blur`, que implementa o filtro de média por meio da função `cv2.blur`, equivalente à convolução da imagem com um *kernel* uniforme cujos coeficientes são $h(x,y) = 1/N^2$; de forma equivalente, o mesmo resultado pode ser obtido com `mm::conv`, calculando $(g = f * h)$. À medida que o *kernel* aumenta, mais pixels contribuem para cada valor de saída, intensificando a suavização, reduzindo o ruído e tornando detalhes finos e bordas progressivamente mais borrados.

```
%%writefile tmp/fig_03_media.cpp
#define MM_OUT "tmp/fig_03_media.png"
// Compile: g++ -std=c++17 -o programa programa.cpp -I. -lstdc++fs

#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]

    // img_gray já está inicializado (inserido automaticamente)

    // 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;
    imgs.push_back(img_gray_crop);

    for (int k : sizes) {
        imgs.push_back(mm::blur(img_gray_crop, k)); // ou
        // mm::Image kernel = mm::Kernel::mean(k);
        // imgs.push_back(mm::conv(img_gray_crop, mm::Kernel::mean(k)));
    }

    std::vector<std::string> titles;
    titles.push_back("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)")
```

Figura 3.12: Filtro de média com *kernels* de tamanho crescente (3×3, 7×7, 15×15). O borramento das bordas aumenta com o tamanho do *kernel*.

3.5.2 Filtro Gaussiano

O filtro Gaussiano pesa os pixels da vizinhança de acordo com uma função Gaussiana bidimensional:

$$G(s,t) = \frac{1}{2\pi\sigma^2}\, e^{-\frac{s^2+t^2}{2\sigma^2}} \tag{3.14}$$

onde σ é o desvio padrão e controla o raio de influência. Pixels mais próximos do centro têm peso maior; pixels distantes são progressivamente ignorados.

A Figura 3.13 apresenta o *kernel* Gaussiano 5×5 gerado para $\sigma = 1$. O *kernel* foi construído a partir do produto externo de dois vetores Gaussianos unidimensionais e posteriormente normalizado para que a soma de seus coeficientes seja igual a 1. Observa-se que os maiores pesos concentram-se no centro da matriz, decrescendo radialmente em direção às bordas. Essa distribuição faz com que os pixels centrais tenham maior influência no resultado da filtragem, contribuindo para uma suavização mais natural e com melhor preservação de bordas do que o filtro de média.

```
%%writefile tmp/fig_03_gauss_kernel.cpp
#define MM_OUT "tmp/fig_03_gauss_kernel.png"
#include "morph.hpp"
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>

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++) {
            if (x > 0) std::cout << "  ";
            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, 40);

    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)")
```

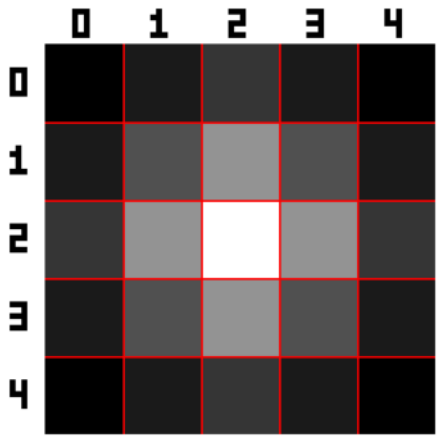


Figura 3.13: *Kernel* Gaussiano 5×5 (=1): resposta ao impulso do filtro — pesos maiores no centro, decrescendo radialmente.

i Vantagem computacional da separabilidade

Considere um *kernel* quadrado de tamanho $n \times n$. Se esse filtro for separável (como o Gaussiano), a convolução 2D pode ser decomposta em duas convoluções 1D: uma horizontal e outra vertical.
Nesse caso, o custo por pixel passa de aproximadamente $O(n^2)$ operações (convolução 2D direta) para $O(2n)$ operações (duas convoluções 1D). Assim, a complexidade é reduzida de forma significativa, tornando o processamento mais eficiente

Comparado ao filtro de média, o Gaussiano:

- **Preserva melhor as bordas** — a ponderação radial suaviza sem criar transições abruptas;
- **Não introduz anéis** (*ringing*) no domínio da frequência, pois a Gaussiana é sua própria transformada de Fourier (Capítulo 5);
- **É controlado por** σ — aumentar σ equivale a aumentar o raio de suavização de forma contínua e previsível.

A Figura 3.14 compara os filtros de média e Gaussiano aplicados à imagem do leopardo usando uma janela 9×9. O filtro de média foi implementado por convolução com um *kernel* uniforme, em que todos os 81 pixels da vizinhança possuem o mesmo peso (1/81), enquanto o filtro Gaussiano foi obtido com `cv2.GaussianBlur`, utilizando pesos definidos por uma distribuição Gaussiana. Ambos reduzem ruído e suavizam a imagem, porém o filtro Gaussiano preserva melhor as bordas e os detalhes locais, como pode ser observado na região ampliada do olho.

A Figura 3.14 compara os filtros de média e Gaussiano aplicados a um detalhe da imagem do leopardo com *kernels* 9×9. O filtro de média foi obtido com `mm::blur`, equivalente à convolução com um *kernel* uniforme cujos coeficientes valem 1/81, enquanto o filtro Gaussiano foi obtido com `mm::gaussian`, equivalente à convolução com um *kernel* gerado a partir de uma distribuição Gaussiana. Ambos promovem suavização e redução de ruído, porém o filtro Gaussiano atribui maior peso aos pixels centrais da vizinhança, preservando melhor as bordas e os detalhes locais, como pode ser observado na região ampliada do olho.

```
%%writefile tmp/fig_03_gauss.cpp
#define MM_OUT "tmp/fig_03_gauss.png"
#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_45.png");
// [pdi:state-io:end]

    // Filtro de média 9×9
    mm::Image img_media9 = mm::blur(img_gray, 9);
    // Filtro Gaussiano 9×9 com  =0 (sigma automático)
    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)")
```

Figura 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 Filtragem Espacial de Realce

Os filtros de realce (*sharpening filters*) enfatizam transições abruptas de intensidade, aumentando a nitidez e a visibilidade de bordas. São filtros **passa-alta** — amplificam as componentes de alta frequência (bordas, textura) e suprimem as de baixa frequência (regiões uniformes).

A intuição é simples: se subtrairmos de uma imagem sua versão suavizada (que contém apenas as baixas frequências), o que resta são as altas frequências — bordas e detalhes. Somando esse resíduo de volta à imagem original, o contraste local aumenta:

$$g = f + k\,(f - f_{\text{suave}}), \quad k > 0 \tag{3.15}$$

O Figura 3.15 ilustra esse processo em um sinal 1D sintético com três estruturas distintas: um degrau largo, um pico fino e uma rampa suave. No painel , o sinal original $f(x)$; no , a versão suavizada $f_{\text{suave}}(x)$ obtida por média móvel — note como o pico fino é atenuado. O painel exibe o resíduo $f - f_{\text{suave}}$, que retém apenas as transições abruptas. Por fim, o painel mostra $g(x)$: o pico, antes atenuado, é restaurado e amplificado em relação ao original. Ajuste k e o tamanho da janela para observar o *trade-off* entre nitidez e amplificação de ruído.

Os filtros de realce formalizam essa ideia diretamente no *kernel*, sem precisar de duas etapas separadas.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap03/sim-03-filtragem1d.html

Figura 3.15: Simulador: Filtragem Espacial de Realce 1D (Unsharp Masking e High-Boost)

3.6.1 Laplaciano

O Laplaciano é um operador de **segunda derivada** isotrópico, ou seja, responde igualmente a variações em todas as direções, ao contrário de operadores de primeira derivada, como Sobel e Prewitt, que são direcionais:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{3.16}$$

Uma propriedade importante da segunda derivada é que seu valor é **próximo de zero em regiões uniformes** e elevado nas transições de intensidade. Assim, ao subtrair o Laplaciano da imagem original, reforçam-se bordas e detalhes, aumentando o contraste local:

$$g(x, y) = f(x, y) - \nabla^2 f(x, y) \tag{3.17}$$

Na forma discreta, a segunda derivada em x é aproximada por $f(x+1, y) - 2f(x, y) + f(x-1, y)$, e analogamente em y. Somando as duas direções, obtém-se o *kernel* w_4 (4-vizinhos) ou w_8 (8-vizinhos, incluindo diagonais):

$$w_4 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \qquad w_8 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.18}$$

Soma zero e centro negativo

Ambos os *kernels* têm **soma dos coeficientes igual a zero**: em regiões uniformes, a saída é 0 — o Laplaciano não altera o brilho médio, apenas detecta variações. O **centro negativo** indica que o pixel é comparado com seus vizinhos: quanto mais ele se destacar (para cima ou para baixo), maior o valor absoluto do Laplaciano naquele ponto.

No exemplo a seguir, o pixel central $[1, 1] = 107$ possui vizinhos $\{95, 106, 108, 103\}$. Como esses valores são próximos entre si, a região é quase uniforme e o Laplaciano retorna um valor baixo, produzindo pouco realce. Em regiões de borda, onde há diferenças maiores entre o pixel central e seus vizinhos, o Laplaciano assume valores mais elevados (positivos ou negativos), e a operação de Equação 3.17 intensifica essas transições.

A Figura 3.16 ilustra o cálculo do Laplaciano com o *kernel* w_4 em uma vizinhança 3×3 destacada dentro de um *patch* 5×5. O exemplo mostra o valor obtido pelo operador e o correspondente pixel realçado na imagem de saída, que passa de 107 para 123.

```
%%writefile tmp/fig_03_laplaciano_patch.cpp
#define MM_OUT "tmp/fig_03_laplaciano_patch.png"
//| 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

#include <iostream>
#include <vector>
#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_45.png");
// [pdi:state-io:end]

    // patch = mm.crop(img_gray, 250, 255, 250, 255)
    mm::Image patch = mm::crop(img_gray, 250, 255, 250, 255);

    // B = np.ones((3, 3), dtype=np.uint8)
    mm::Kernel B(3, 3, 1.0);

    // print("Patch 5x5 (intensidades):")
    std::cout << "Patch 5x5 (intensidades):\n";

    // print(mm.drawImg(patch))
    std::cout << mm::drawImg(patch) << "\n";

    // mm.drawImgKernel(patch, B, 1, 1, 40)
    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)")
```

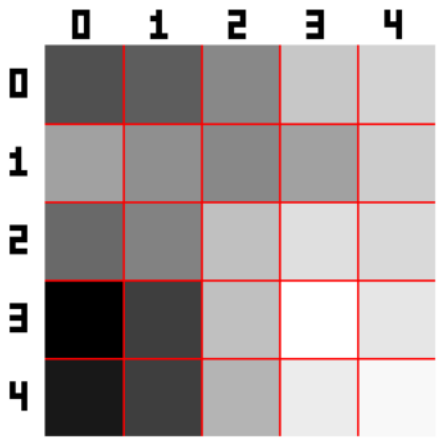


Figura 3.16: *Kernel* Laplaciano w4 sobre o *patch* 5×5: a janela amarela destaca a vizinhança 3×3 onde o operador de segunda derivada é calculado.

A Figura 3.17 compara a aplicação dos *kernels* Laplacianos w_4 e w_8 em um recorte maior da imagem do leopardo. Para cada caso, são mostradas a resposta bruta do operador, que evidencia as bordas e transições de intensidade, e a imagem obtida após o realce por subtração do Laplaciano. Observa-se que o *kernel* w_8, por considerar também os vizinhos diagonais, produz uma resposta mais intensa e detecta variações em mais direções, resultando em um realce ligeiramente mais acentuado.

```
%%writefile tmp/fig_03_laplaciano.cpp
#define MM_OUT "tmp/fig_03_laplaciano.png"
#include "morph.hpp"
#include <iostream>
#include <vector>

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]

    // As variáveis img_gray_crop já estão inicializadas aqui

    // Kernel Laplaciano w4 (vizinhança 4)
    mm::Kernel w4{{0, 1, 0},
                  {1, -4, 1},
                  {0, 1, 0}};

    // Kernel Laplaciano w8 (vizinhança 8)
    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,
        {"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)")
```

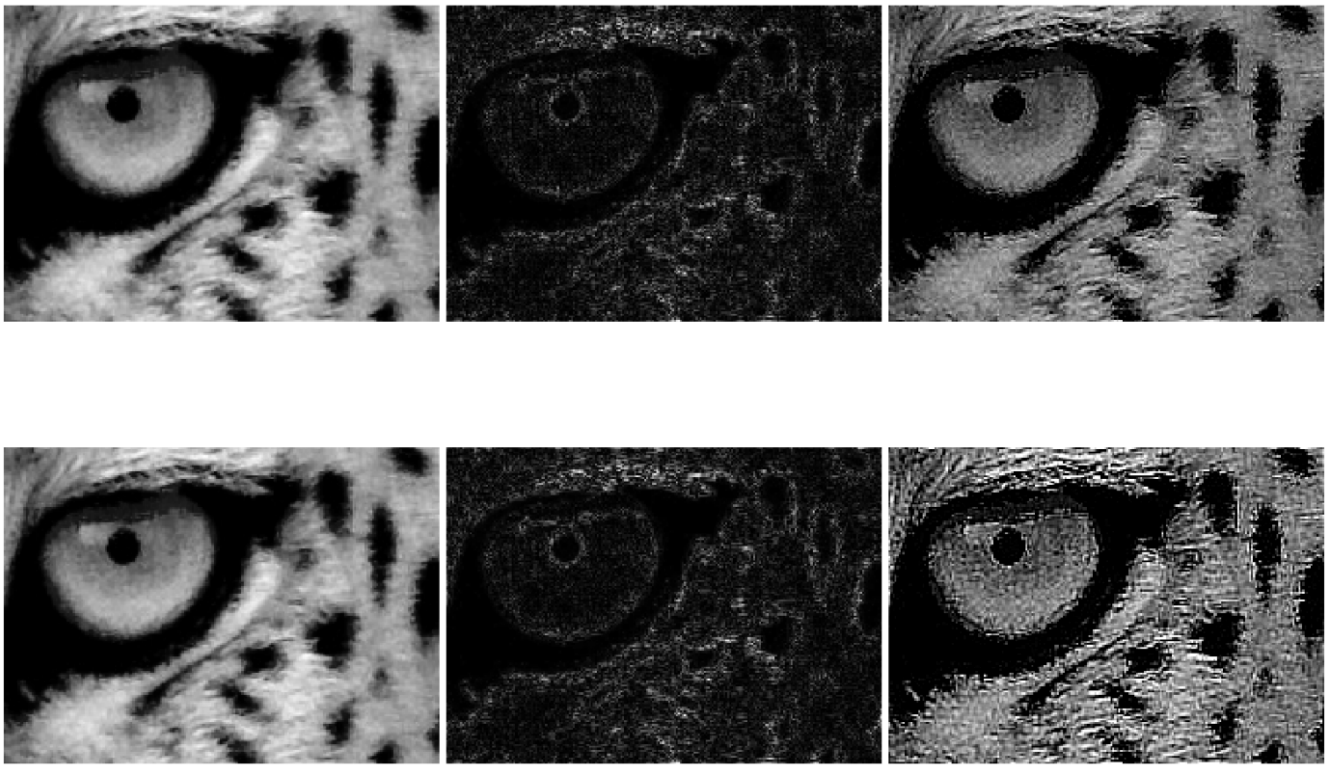


Figura 3.17: Laplaciano aplicado à imagem do leopardo: resposta bruta (bordas) com w4 e w8, e imagens realçadas pela subtração do Laplaciano. w8 é mais sensível às diagonais.

3.6.2 Operador de Sobel

O operador de Sobel estima as **derivadas parciais de primeira ordem** nas direções horizontal e vertical. Diferente do Laplaciano (segunda derivada), o Sobel é direcional e mais robusto ao ruído, pois cada *kernel* combina uma derivada com uma suavização Gaussiana perpendicular:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * f, \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * f \tag{3.19}$$

G_x detecta bordas **verticais** (variação na direção x); G_y detecta bordas **horizontais** (variação na direção y). Os pesos $\{1, 2, 1\}$ na direção perpendicular correspondem à suavização Gaussiana 1D, que reduz a sensibilidade ao ruído.

> **i** Sobel é correlação, não convolução
>
> Os *kernels* de Sobel são **assimétricos** — a rotação de 180° altera o resultado. `cv2.Sobel` implementa correlação cruzada (como `cv2.filter2D`). Para obter a derivada direcional correta, os sinais já estão definidos para correlação: G_x retorna valores positivos onde a intensidade cresce da esquerda para a direita.

A magnitude do **gradiente** combina os dois componentes, representando a força da borda independente de direção:

$$|\nabla f| = \sqrt{G_x^2 + G_y^2} \tag{3.20}$$

E a **direção** do gradiente (perpendicular à borda) é:

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \tag{3.21}$$

Para ilustrar numericamente, calcula-se G_x e G_y manualmente no pixel central $[1, 1]$ do *patch* 5×5:

O valor reduzido de $|\nabla f|$ nesse *patch* confirma que a região é quase uniforme, pois o gradiente assume valores elevados apenas onde há mudanças significativas de intensidade. A Figura 3.18 aplica o operador de Sobel a um recorte maior da imagem do leopardo. São apresentadas as respostas horizontal (G_x) e vertical (G_y), obtidas por convolução com os respectivos *kernels* de Sobel, além da magnitude $|\nabla f|$, calculada a partir da combinação de ambas. Enquanto G_x destaca bordas verticais e G_y bordas horizontais, a magnitude evidencia bordas em qualquer direção.

```
%%writefile tmp/fig_03_sobel.cpp
#define MM_OUT "tmp/fig_03_sobel.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]

    // 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.
    mm::Image mag = mm::sobel(img_gray_crop);

    mm::show({img_gray_crop, mag}, MM_OUT,
             {"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)")
```

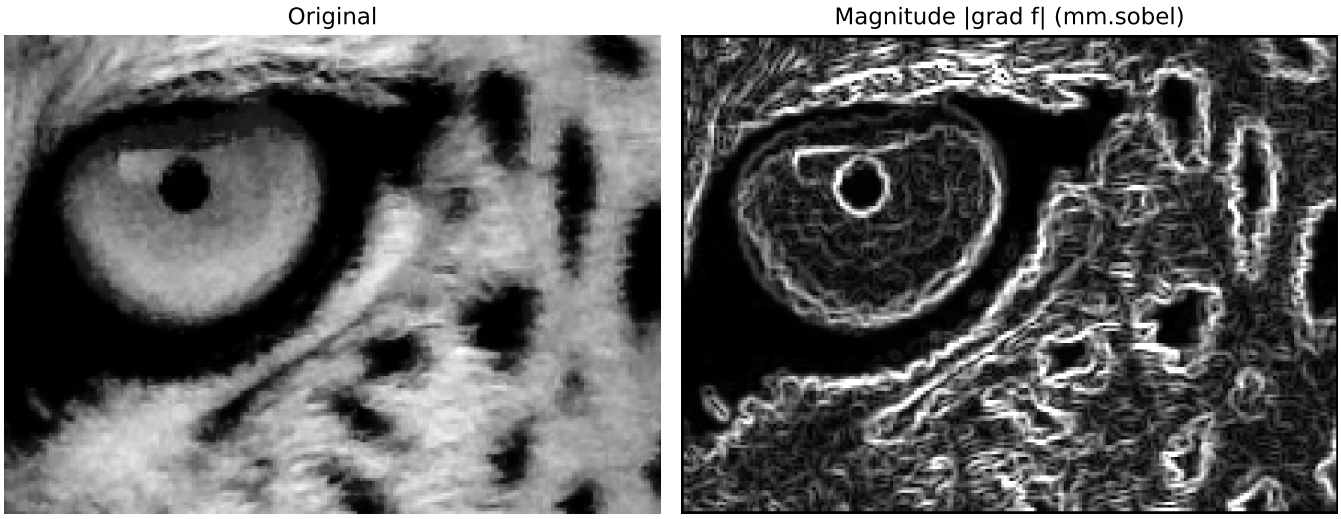


Figura 3.18: Operador de Sobel na imagem do leopardo: a magnitude $|f|$ combina os gradientes horizontal e vertical, revelando todas as bordas. A decomposição Gx/Gy com sinal fica na trilha Python — mm::sobel devolve a magnitude já com clip.

3.6.3 Operador de Prewitt

O operador de Prewitt é estruturalmente idêntico ao Sobel, mas substitui a ponderação Gaussiana $\{1, 2, 1\}$ por pesos uniformes $\{1, 1, 1\}$:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} * f, \qquad G_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} * f \tag{3.22}$$

A magnitude e a direção do gradiente seguem as mesmas equações do Sobel (Equação 3.20 e Equação 3.21). A diferença prática é que o Prewitt é ligeiramente mais sensível ao ruído — a suavização perpendicular uniforme pondera menos o pixel central da linha — mas computacionalmente mais simples. Em imagens com baixo ruído os resultados são equivalentes.

```
%%writefile tmp/fig_03_prewitt.cpp
#define MM_OUT "tmp/fig_03_prewitt.png"
// Compile: g++ -std=c++17 -o program program.cpp -I. -L. -lmorph

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

```cpp
#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]

    // Pré-condição: img_gray_crop já está inicializado como mm::Image
    // (inserido automaticamente)

    // | label: fig-03-prewitt
    // | fig-cap: "Operador de Prewitt: magnitude do gradiente, comparável ao Sobel mas sem a
    ponderação central. mm::prewitt devolve |f| com clip."
    // | echo: true
    // | output: true

    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)")
```

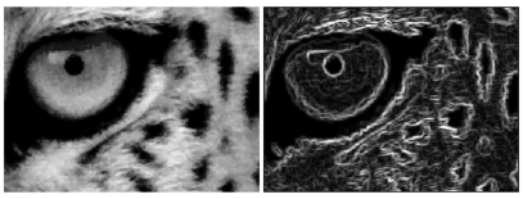


Figura 3.19: Operador de Prewitt: magnitude do gradiente, comparável ao Sobel mas sem a ponderação central. mm::prewitt devolve |f| com clip.

3.6.4 *Unsharp Masking* (USM)

O *Unsharp Masking* é uma técnica clássica de realce de nitidez originária da fotografia analógica, hoje amplamente usada em software de edição de imagens. A ideia central é extrair as **componentes de alta**

frequência da imagem (bordas e detalhes) e somá-las de volta à original com um peso k:

Tabela 3.3: Etapas do *Unsharp Masking*.

Etapa	Operação	Descrição
1	$\bar{f} = f * G_\sigma$	Suaviza com Gaussiana — retém baixas frequências
2	$m = f - \bar{f}$	Máscara: diferença = altas frequências (bordas)
3	$g = f + k \cdot m$	Soma ponderada da máscara à original

Substituindo a etapa 2 na etapa 3, obtém-se a expressão compacta:

$$g = f + k\,(f - f * G_\sigma) = (1 + k)\,f - k\,(f * G_\sigma) \tag{3.23}$$

O parâmetro k controla a intensidade do realce:

- $k = 0$: sem realce ($g = f$);
- $k = 1$: USM clássico — duplica a contribuição das altas frequências;
- $k > 1$: *High Boost Filtering* — amplificação além do dobro, útil para imagens muito borradas.

> ⚠ Amplificação de ruído
>
> O USM não distingue bordas de ruído — ambos são componentes de alta frequência. Para k elevado, o ruído presente na imagem é amplificado junto com as bordas. Por isso, é recomendável aplicar uma leve suavização antes do USM em imagens ruidosas, ou usar σ pequeno na Gaussiana.

Para ilustrar as etapas do USM, a Figura 3.20 aplica o método a um *patch* 30×30 da imagem do leopardo, utilizando $\sigma = 1$ e $k = 1$. Inicialmente, a imagem é suavizada por um filtro Gaussiano. Em seguida, a máscara de alta frequência é obtida pela diferença entre a imagem original e a suavizada. Por fim, essa máscara é somada à imagem original, reforçando bordas e detalhes. A figura apresenta as três etapas do processo e o resultado final do realce.

```
%%writefile tmp/fig_03_usm2.cpp
#define MM_OUT "tmp/fig_03_usm2.png"
//| 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

#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]

    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)")
```

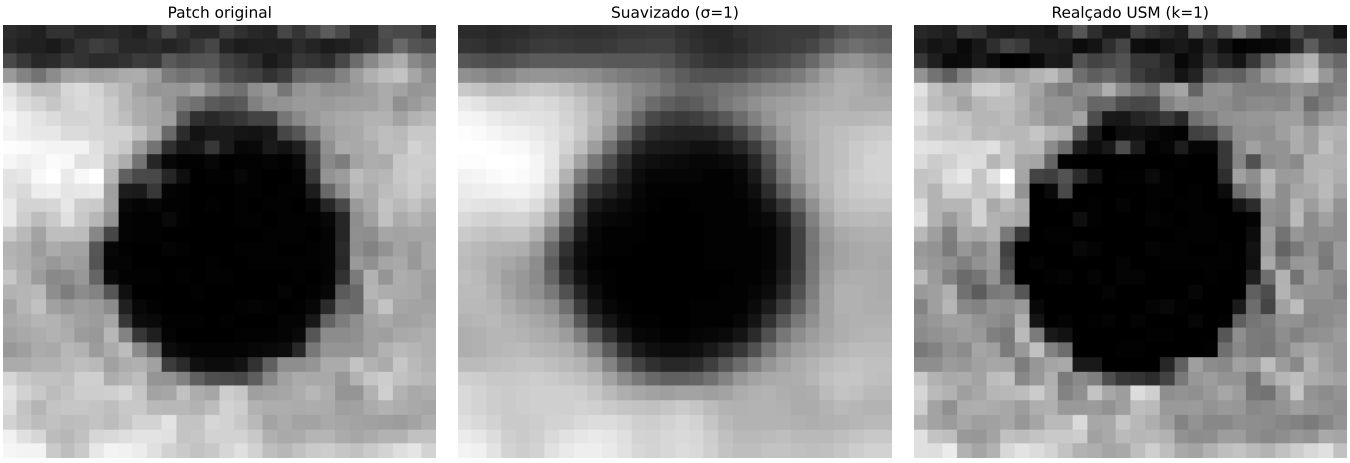


Figura 3.20: Realce por *Unsharp Masking* num *patch* do leopardo: mm::usm faz suavização Gaussiana, subtrai da original (máscara de alta frequência) e reintroduz a máscara realçada.

A Figura 3.21 aplica o método USM a um recorte maior da imagem do leopardo utilizando $\sigma = 1$ e diferentes valores do fator de ganho k. Em todos os casos, a máscara de alta frequência é obtida pela diferença entre a imagem original e sua versão suavizada por filtro Gaussiano. O parâmetro k controla a intensidade do realce: valores menores produzem um aumento sutil de nitidez, enquanto valores maiores reforçam progressivamente bordas e detalhes. Observa-se que, para valores elevados de k, surgem halos ao redor das bordas e o ruído presente na imagem passa a ser amplificado.

```
%%writefile tmp/fig_03_usm.cpp
#define MM_OUT "tmp/fig_03_usm.png"
#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]

    // Imagem original e versões com Unsharp Masking com diferentes valores de k
    mm::show(
        {img_gray_crop,
         mm::usm(img_gray_crop, 0.5), mm::usm(img_gray_crop, 1.0),
         mm::usm(img_gray_crop, 3.0), mm::usm(img_gray_crop, 5.0),
         mm::usm(img_gray_crop, 8.0)},
        MM_OUT,
        {"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)")
```

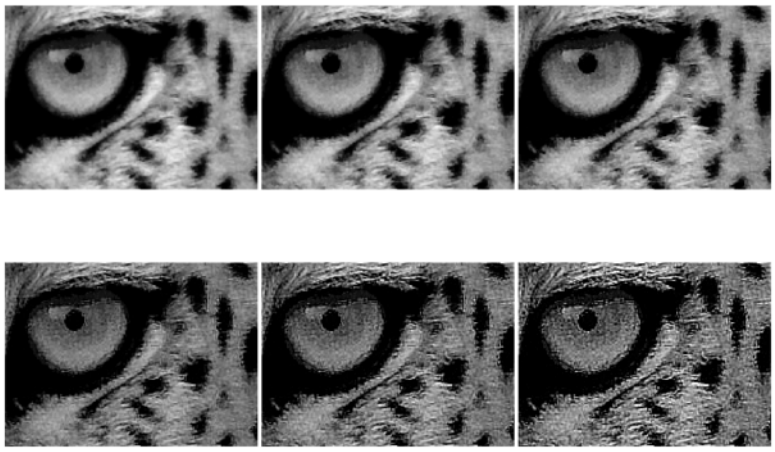


Figura 3.21: *Unsharp Masking* na imagem do leopardo com =1 e k de 0.5 a 8.0. Para k>2 surgem halos nas bordas e o ruído de fundo aparece.

3.6.5 Detector de Canny

O Canny combina quatro etapas em sequência — suavização Gaussiana, gradiente de Sobel, supressão de não-máximos e histerese por duplo limiar — para produzir bordas **finas, binárias e conectadas**. Diferente de Sobel e Prewitt, o resultado não é um mapa de gradiente contínuo, mas uma máscara onde cada pixel é borda ou não.

O parâmetro central é o par de limiares (T_{low}, T_{high}). Pixels com gradiente acima de T_{high} são bordas certas; abaixo de T_{low}, descartados. Os pixels ambíguos — entre os dois limiares — são decididos por **histerese**: tornam-se borda se estiverem conectados a uma borda certa, e descartados caso contrário. Isso evita tanto a perda de trechos fracos de bordas reais quanto a inclusão de ruído isolado. Uma heurística comum é $T_{high} = 3 \times T_{low}$.

```
%%writefile tmp/fig_03_canny.cpp
#define MM_OUT "tmp/fig_03_canny.png"
#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]
```

```
    // img_gray_crop já está inicializado pela linha inserida automaticamente

    //| 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

    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)")
```

Figura 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.

Escolha dos limiares

Uma heurística comum é $T_{high} = 3 \times T_{low}$. Valores típicos dependem do intervalo de gradiente da imagem — `cv2.Canny` aceita valores absolutos em $[0, 255]$. Para imagens com contraste variável, calcular os limiares a partir de percentis da magnitude de Sobel é mais robusto do que valores fixos.

3.7 Filtros de Ordem: Filtro da Mediana

Os filtros de ordem (*order-statistic filters*) substituem o pixel central pelo valor de um **percentil** da distribuição de intensidades da vizinhança — ao contrário dos filtros lineares, que calculam combinações ponderadas. O mais importante é o **filtro da mediana**.

3.7.1 Ruído Impulsivo: Sal e Pimenta

O ruído **sal e pimenta** (*salt-and-pepper noise*) substitui pixels aleatórios por valores extremos: 0 (pimenta, preto) ou 255 (sal, branco). É comum em transmissão de imagens com erros de bit e em câmeras com sensores defeituosos.

Para entender por que os filtros lineares falham, considere uma vizinhança 3×3 onde um único pixel foi corrompido para 255:

$$\text{vizinhança} = \begin{bmatrix} 102 & 98 & 105 \\ 100 & \mathbf{255} & 97 \\ 103 & 99 & 101 \end{bmatrix}$$

Tabela 3.4: Média vs. mediana com um pixel corrompido. A mediana ignora o outlier; a média é deslocada ~40 níveis.

Método	Cálculo	Resultado
Média	(102+98+…+255+…+101)/9	140
Mediana	{97,98,99,100,**101**,102,103,105,255}	101

Por que filtros de média falham com ruído impulsivo?

A média é sensível a ***outliers*** — um único pixel com valor 255 em uma vizinhança de valor 100 eleva a saída para 140, espalhando o ruído pela imagem. A mediana, por ser um **estimador robusto**, seleciona o valor central da distribuição ordenada, descartando naturalmente os extremos sem nenhum ajuste especial.

O exemplo a seguir ilustra o comportamento da média e da mediana na presença de um pixel corrompido por ruído impulsivo. Observa-se que a média é fortemente influenciada pelo valor extremo (255), produzindo uma estimativa distante dos valores predominantes da vizinhança. Já a mediana permanece próxima do valor original da região, evidenciando sua maior robustez a *outliers* e justificando seu uso na remoção de ruído sal e pimenta.

A Figura 3.23 apresenta o efeito do ruído sal e pimenta em diferentes densidades. O ruído foi gerado substituindo aleatoriamente uma fração dos pixels por valores mínimos (0, pimenta) e máximos (255, sal). À medida que a densidade aumenta de 2% para 10%, cresce a quantidade de pixels corrompidos, tornando a degradação visual mais evidente e dificultando a percepção de detalhes da imagem.

```
%%writefile tmp/fig_03_ruido.cpp
#define MM_OUT "tmp/fig_03_ruido.png"
#include "morph.hpp"
```

```cpp
#include <iostream>
#include <random>
#include <filesystem>

// Função para adicionar ruído sal e pimenta
mm::Image salt_pepper(const mm::Image& img, double prob) {
    mm::Image out = img;  // cópia da imagem
    int h = out.h;
    int w = out.w;

    std::mt19937 rng(42);  // gerador com semente fixa
    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_prob(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);
        // 50% de chance de ser sal (255) ou pimenta (0)
        out.at(y, x) = (dist_prob(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]

    // As variáveis img_gray_crop já estão inicializadas aqui

    // Aplica ruído sal e pimenta com diferentes densidades
    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);

    // Exibe as imagens
    mm::show(std::vector<mm::Image>{img_gray_crop, n2, n5, n10},
             MM_OUT,
             std::vector<std::string>{"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)")
```

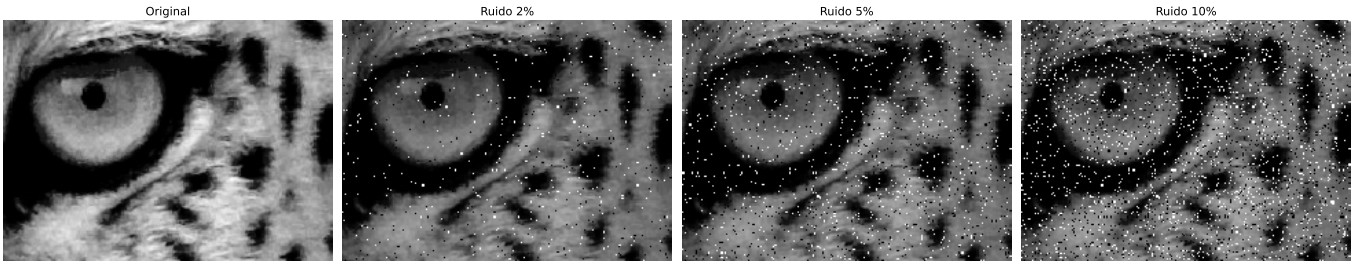


Figura 3.23: Ruído sal e pimenta com densidades crescentes (2%, 5%, 10%): metade dos pixels corrompidos vira sal (255), metade pimenta (0).

3.7.2 Filtro da Mediana

O filtro da mediana substitui cada pixel pelo **valor mediano** dos pixels da sua vizinhança $n \times n$:

$$g(x,y) = \text{med}_{(s,t)\in\mathcal{V}_n}\{f(x+s, y+t)\} \tag{3.24}$$

O valor mediano é aquele que ocupa a posição central quando os n^2 valores da vizinhança são ordenados. Para uma janela 3×3 ($n^2 = 9$ pixels), a mediana é o 5^{o} valor da sequência ordenada.

Para ilustrar, considere o mesmo *patch* 5×5 com um pixel corrompido artificialmente em $[1, 1]$:

O exemplo confirma: mesmo com o pixel corrompido a 255, a mediana retorna o valor central correto — o *outlier* ocupa a última posição na ordenação e é descartado naturalmente.

Por ser baseada em ordenação e não em soma, a mediana possui três propriedades fundamentais que a diferenciam dos filtros lineares:

- **Robusta** ao ruído impulsivo — outliers vão para as extremidades da sequência ordenada e não afetam o valor central;

- **Preservadora de bordas** — transições abruptas de intensidade são mantidas, pois a mediana seleciona um valor que já existe na vizinhança, sem criar novos níveis intermediários;
- **Não linear** — não pode ser expressa como convolução, portanto `mm::conv` não se aplica; usa-se `cv2.medianBlur`.

A Figura 3.24 compara diferentes técnicas de remoção de ruído sal e pimenta aplicadas a uma imagem com 10% de pixels corrompidos. Foram avaliados os filtros Gaussiano, Média, Mediana, Bilateral e Morfológico (próximo capítulo), permitindo observar o compromisso entre remoção de ruído e preservação de detalhes. Em geral, os filtros de média e Gaussiano reduzem o ruído, mas tendem a borrar as bordas, enquanto a mediana apresenta melhor desempenho para ruído impulsivo. O filtro bilateral preserva melhor as bordas, e o filtro morfológico remove boa parte dos pixels corrompidos sem degradar excessivamente a estrutura da imagem.

```
%%writefile tmp/fig_03_ruido_filtros.cpp
#define MM_OUT "tmp/fig_03_ruido_filtros.png"
#include "morph.hpp"
#include <random>
#include <vector>
#include <string>
#include <filesystem>

// Função para adicionar ruído sal e pimenta
mm::Image salt_pepper(const mm::Image& img, double prob) {
    mm::Image out = img;  // cópia
    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_prob(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_prob(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]

    // img_gray_crop já está inicializado

    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)")
```

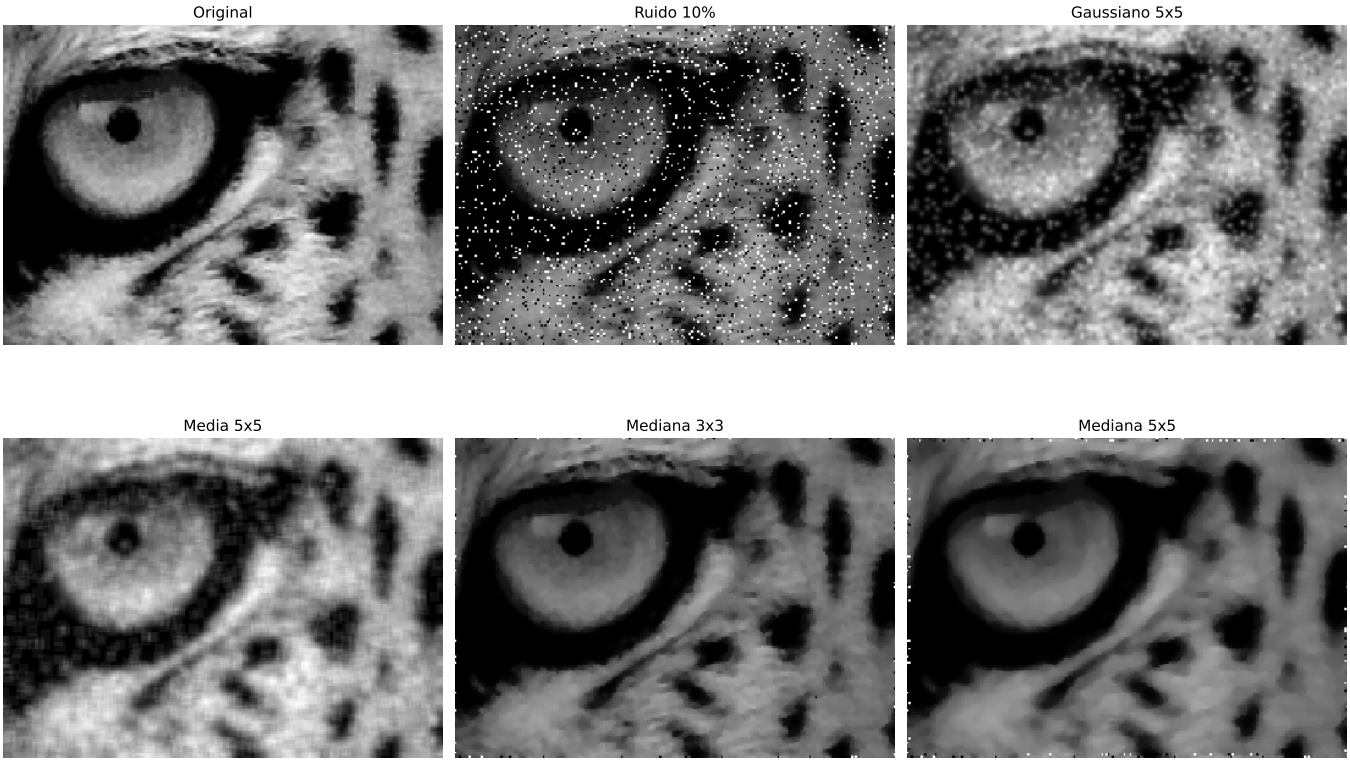


Figura 3.24: Filtros para ruído sal e pimenta (10%): Gaussiano, Média e Mediana. Bilateral e morfológico (open+close) ficam só na trilha Python — bilateral não tem equivalente em morph.hpp e morfologia é do próximo capítulo.

3.8 Aplicação Prática: Pré-processamento para Segmentação

Na prática, as técnicas deste capítulo raramente são usadas isoladamente. Um ***pipeline* de pré-processamento** típico combina várias etapas em sequência, adaptando-se ao tipo de imagem e à aplicação. A Figura 3.25 ilustra um *pipeline* completo:

1. **Equalização de histograma (CLAHE):** normaliza o contraste independente das condições de iluminação;
2. **Filtro Gaussiano:** suaviza ruído de aquisição sem destruir bordas;
3. **Detecção de bordas (Sobel/Canny):** extrai estruturas relevantes para segmentação.

i Ordem importa

A ordem das operações afeta o resultado final. Em geral: **(1) normalização de intensidade → (2) redução de ruído → (3) realce/segmentação**. Inverter a ordem pode amplificar ruído ou perder bordas antes de detectá-las.

```
%%writefile tmp/fig_03_pipeline.cpp
#define MM_OUT "tmp/fig_03_pipeline.png"
// Compile: g++ -std=c++17 -o pipeline pipeline.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_crop = mm::_read_state("tmp/state/img_gray_crop_58.png");
// [pdi:state-io:end]

    // img_gray_crop já está inicializado aqui

    mm::Image img_eq = mm::equalize(img_gray_crop);    // Etapa 1: equalização global
    mm::Image img_gauss = mm::gaussian(img_eq, 5, 0);  // Etapa 2: Gaussiano
    mm::Image edges = mm::canny(img_gauss, 50, 150);   // Etapa 3: Canny

    mm::Image edges_direct = mm::canny(img_gray_crop, 50, 150);  // Canny direto, sem
pré-processo

    mm::show(
        std::vector<mm::Image>{img_gray_crop, img_eq, img_gauss, edges, edges_direct},
        MM_OUT,
        std::vector<std::string>{"Original", "1. Equalizado", "2. Gaussiano", "3. Canny
(pipeline)", "Canny (direto)"},
        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. Equalizado
[3] 2. Gaussiano
[4] 3. Canny (pipeline)
[5] Canny (direto)
```

```
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)")
```

Figura 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 Resumo

Neste capítulo foram apresentadas as principais técnicas de processamento no domínio espacial, da manipulação direta de pixels até a filtragem por vizinhança:

- **Operações de ponto:** aritméticas saturadas (`mm::addm`, `mm::subm`) e lógicas bit a bit (`mm::band`, `mm::bor`, `mm::bnot`) para recorte de ROI e combinação de imagens; *alpha blending* (`mm::blend`) para fusão ponderada com peso $\alpha \in [0, 1]$.
- **Histograma:** função discreta de distribuição de intensidades; visualizado com `mm::histImg` e calculado com `mm::hist`; base para diagnóstico tonal e para as técnicas de equalização e especificação.
- **Equalização:** redistribuição automática das intensidades pela CDF (`mm::equalize`), com variante adaptativa CLAHE para controle local do contraste.
- **Especificação de histograma:** transferência do perfil tonal de uma imagem de referência via mapeamento inverso da CDF — generalização da equalização para distribuições arbitrárias.
- **Correlação e convolução:** mecanismo de janela deslizante implementado em `mm::conv` (`cv2.filter2D`); diferenciados pela rotação de 180° do *kernel* — relevante apenas para kernels assimétricos.
- **Filtros de suavização:** média (*kernel* uniforme, borra bordas proporcionalmente ao tamanho) e Gaussiano (ponderação radial, separável, sem *ringing*, preserva melhor as bordas).
- **Filtros de realce:** Laplaciano (w_4/w_8, segunda derivada isotrópica), Sobel (gradiente direcional de primeira ordem, com magnitude $|\nabla f|$ e direção θ) e *Unsharp Masking* (amplificação das altas frequências com parâmetro k).
- **Filtro da mediana:** não linear, robusto a outliers, preserva bordas — superior aos filtros lineares para ruído sal e pimenta.
- **Pipeline prático:** encadeamento CLAHE → Gaussiano → Canny como estratégia de pré-processamento; `mm::drawImgKernel` para visualização didática da janela deslizante.

O Capítulo 4 abordará a **morfologia matemática** (erosão, dilatação, abertura e fechamento), explorando em profundidade as funções `mm::ero` e `mm::dil` da biblioteca `morph.py`. Em seguida, o Capítulo 5 apresentará o **processamento no domínio da frequência**, com foco na Transformada de Fourier e em técnicas de filtragem espectral.

3.10 🤖 Uso do Gemini Notebook como Tutor Complementar

Nesta edição, incentivamos o uso do **Gemini Notebook** como ferramenta complementar de aprendizagem. Essa ferramenta de IA utiliza exclusivamente os documentos fornecidos pelo autor como base de conhecimento,

garantindo respostas coerentes com o conteúdo do livro — incluindo as funções da biblioteca `morph.py` e os experimentos realizados neste capítulo.

Para cada capítulo, preparamos um projeto específico na plataforma com o PDF do capítulo, os *notebooks* e materiais auxiliares. Sugerimos explorar especialmente:

- **Guia de Estudo:** resumo estruturado dos conceitos, ideal para revisão antes de provas;
- **Conversa:** tire dúvidas sobre equalização, convolução, filtros e pipelines diretamente com o tutor;
- **Perguntas frequentes:** questões típicas sobre a diferença entre média e mediana, USM, Laplaciano vs. Sobel.

🎓 Estude com o Tutor Inteligente

Para interagir com o conteúdo deste capítulo, acesse o *link* a seguir. O ambiente contém materiais didáticos em diferentes formatos, gerados a partir do **PDF** do capítulo. Na plataforma, explore especialmente as opções **Guia de Estudo** e **Conversa** para aprofundar sua compreensão.
🚀 ACESSAR Gemini Notebook: CAPÍTULO 03

🌐 Idioma e Linguagem de Programação

O projeto deste capítulo no Gemini Notebook foi construído apenas com o texto em **português** e os exemplos de código em **Python**. Se você está estudando pela edição em inglês ou francês, ou acompanhando a trilha em C++, as respostas do tutor podem não corresponder exatamente à versão que você está lendo.

⚠️ Aviso sobre Conteúdo Gerado por IA

A IA é uma poderosa aliada nos estudos, mas o conteúdo gerado pode conter **erros ou imprecisões**. Consulte sempre **livros, artigos científicos e outras fontes acadêmicas confiáveis** para validar as informações. Sempre que possível, execute os exemplos práticos fornecidos neste capítulo para verificar os resultados.

3.11 Lista de Exercícios

1. **(10%)** Explique a diferença entre **convolução** e **correlação cruzada**. Para quais tipos de *kernel* os resultados são idênticos? Dê um exemplo de *kernel* assimétrico (como Sobel G_x) e mostre numericamente que os resultados diferem aplicando-o ao patch 5×5 do capítulo das duas formas.

2. **(15%)** Considere uma imagem 5×5 com intensidades concentradas entre os níveis 3 e 5 (baixo contraste, 3 bits). Aplique manualmente o algoritmo de equalização da Tabela 3.1, preenchendo todas as colunas da tabela (k, $h[k]$, $p[k]$, cdf$[k]$, lut$[k]$). Verifique o resultado com `mm::equalize`.

3. **(15%)** Usando `mm::conv`, aplique o filtro de média com kernels de tamanho 3×3, 9×9 e 21×21 à imagem do mandrill. Para cada versão, calcule o **PSNR** (*Peak Signal-to-Noise Ratio*) em relação à original:

$$\text{PSNR} = 10\log_{10}\left(\frac{255^2}{\text{MSE}}\right), \quad \text{MSE} = \frac{1}{MN}\sum_{i,j}(f-g)^2$$

   Plote o PSNR em função do tamanho do kernel e explique o que a queda progressiva indica sobre a relação entre suavização e perda de informação.

4. **(15%)** Usando `add_salt_pepper` com densidade de 5%, aplique e compare: (a) `mm::conv` com média 3×3, (b) `cv2.GaussianBlur` com $\sigma = 1$, (c) `cv2.medianBlur` com janela 3×3 e (d) `cv2.medianBlur` com janela 5×5. Exiba as imagens com `mm::show` em grade 2×4 (linha 1: imagens, linha 2: histogramas via `mm::histImg`). Explique por que a mediana supera os filtros lineares usando o argumento da Tabela 3.4.

5. **(15%)** Implemente `mm::conv0` usando apenas operações NumPy vetorizadas — sem laços Python e sem `cv2.filter2D` — com o operador de *stride tricks* (`np.lib.stride_tricks.sliding_window_view`). Compare o resultado e o tempo de execução com `mm::conv0` (laços) e `mm::conv` (cv2) para kernels 3×3 e 15×15 na imagem do mandrill.
6. **(15%)** Aplique o *Unsharp Masking* com $\sigma = 1$ e $k \in \{0.5, 1.0, 2.0, 4.0\}$ usando a função `usm` do capítulo. Para cada valor de k: (a) calcule a diferença absoluta $|g - f|$, (b) exiba as imagens e as diferenças com `mm::show`, e (c) plote o histograma das diferenças com `mm::histImg`. Identifique a partir de qual k os artefatos (halos e amplificação de ruído) tornam-se visualmente inaceitáveis.
7. **(15%)** Escolha uma imagem de raio-X ou tomografia disponível publicamente (ex.: via `mm::read` de URL) e projete um pipeline de pré-processamento com pelo menos 4 etapas sequenciais, justificando cada escolha com base nos conceitos do capítulo. Exiba com `mm::show` em grade: imagem original, cada etapa intermediária e o resultado final com seus histogramas (`mm::histImg`).

Referências do Capítulo

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

- Gonzalez (2018) para os conceitos de operações de intensidade, histograma, convolução e filtragem espacial.
- Szeliski (2022) para a visão computacional e aplicações práticas de filtragem.
- Bradski (2008) para a implementação prática com OpenCV e `morph.py`.

3.12 💻 Parte Prática com Exercícios de Programação

🎯 Objetivo deste Caderno

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

Download

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

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # artefatos de build da trilha C++ (.cpp, binário,
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")

# O kernel é Python mesmo na trilha C++: `mm` (morph.py) é usado pelos
# simuladores, pela exibição das figuras que o binário C++ gera e pelo
# estado mm::Image entre células. cpp=True baixa também a trilha compilada
# (morph.hpp + stb_image*.h), usada no #include das células %%writefile *.cpp.
import config
config.setup(testsuite=True, cpp=True)
from morph import mm
from testsuite import TestSuite
```

```
Ambiente pronto. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Executando os Testes

Para avaliar os testes, execute `TestSuite("EP03_01.extensão").run()` numa nova célula, trocando a extensão pela da linguagem usada (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). O sistema baixa os casos de teste do GitHub, executa o programa e calcula a nota automaticamente.

Para testar código Python diretamente, sem salvar arquivo, use `run_code(codigo)` passando o código como *string* numa variável `codigo`:

```
codigo = """
from morph import mm
# ... seu código aqui ...
"""
TestSuite("EP03_01").run_code(codigo)
```

3.12.1 EP03_01 ➕ Adição Saturada de Constante

Em sistemas de vigilância por vídeo, câmeras em ambientes com iluminação variável produzem imagens subexpostas. O ajuste de brilho por **adição saturada de uma constante** é a operação mais simples para correção imediata, sendo aplicada em tempo real nos *chips* de câmeras embarcadas e em *pipelines* de pré-processamento de robôs móveis.

Ver na Figura 3.26 uma simulação deste EP.

3.12.1.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros L (linhas) e C (colunas).
2. **Constante:** Ler o inteiro k (valor a ser somado).
3. **Dados:** Ler os valores inteiros da matriz original linha a linha.
4. **Mapeamento:** Para cada pixel p, calcular o novo valor pela equação:

$$p' = \text{clip}(p + k)$$

5. **Saída:** Exibir a matriz resultante com dimensões $L \times C$.

3.12.1.2 📌 Restrições Computacionais

- **Saturação (*Clipping*):** Os valores devem ser confinados ao intervalo $[0, 255]$:

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

- **Tipo:** O resultado final deve ser inteiro (sem casas decimais).
- k **pode ser negativo:** valores negativos escurecem a imagem; positivos clareiam.

3.12.1.3 🧠 Fundamentação Teórica

Parâmetro	Tipo	Impacto Visual
$k > 0$	Inteiro	Clareia a imagem; pixels próximos de 255 saturam em branco
$k < 0$	Inteiro	Escurece a imagem; pixels próximos de 0 saturam em preto
$k = 0$	Inteiro	Imagem inalterada

3.12.1.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

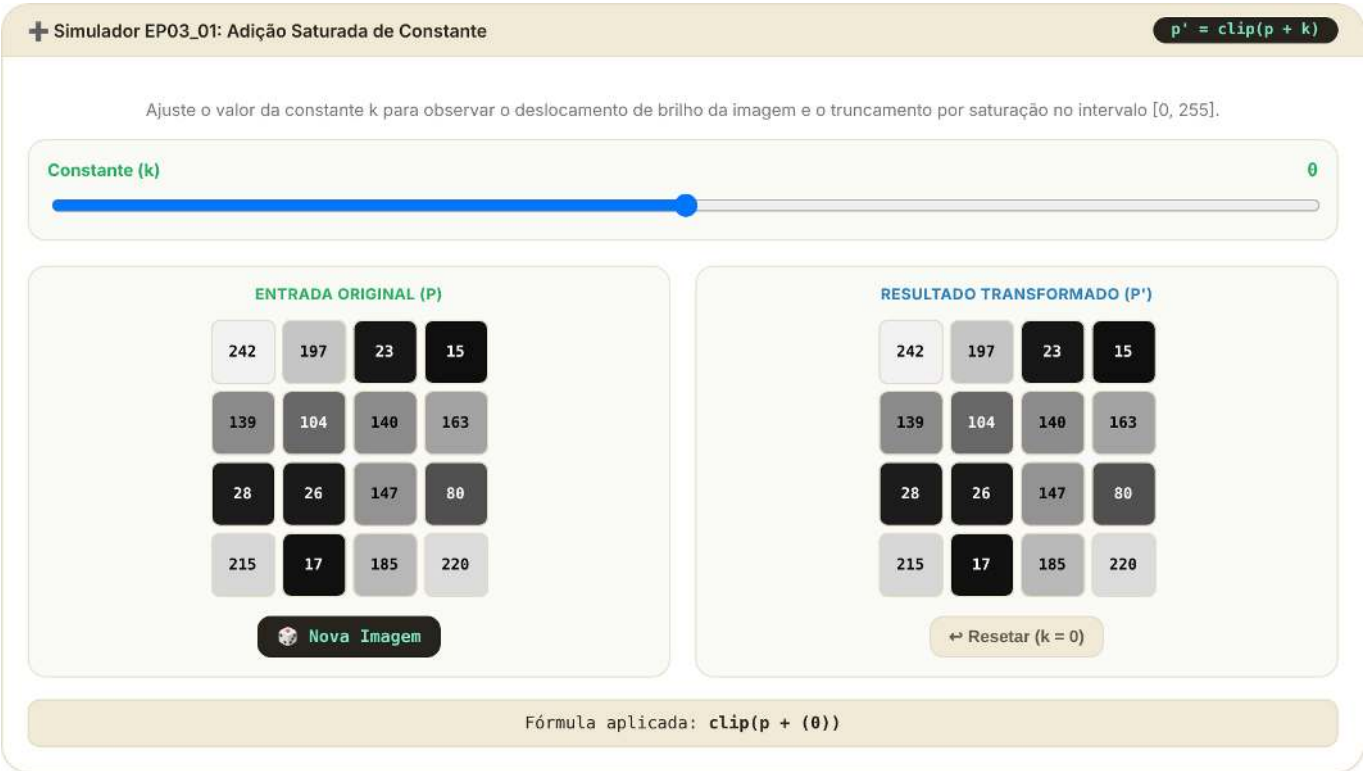
- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linha 3: Inteiro k.
- Linhas seguintes: Elementos inteiros da matriz original.

Saída:

- Matriz transformada em L linhas e C colunas, valores inteiros separados por espaço.

3.12.1.5 📌 Exemplos

Entrada	Saída	Observação
2 3 50 0 100 200 210 240 255	50 150 250 255 255 255	Saturação em 255 nos pixels altos
1 4 -30 0 20 200 255	0 0 170 225	Saturação em 0 nos pixels baixos



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap03/sim-ep0301-adicao.html

Figura 3.26: Simulador EP03_01: Adição Saturada de Constante (p' = clip(p + k))

```
%%writefile EP03_01.cpp
// sua solução
```

```
Overwriting EP03_01.cpp
```

```
TestSuite("EP03_01.cpp").run()
```

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

3.12.2 EP03_02 🔀 Alpha *Blending* de Duas Imagens

Em medicina nuclear, imagens de diferentes modalidades (tomografia computadorizada e ressonância magnética) são fundidas para auxiliar no diagnóstico. A **mistura ponderada** (*alpha blending*) é a operação fundamental desse processo, permitindo ao radiologista controlar interativamente o peso de cada modalidade na imagem exibida.

Ver na Figura 3.27 uma simulação deste EP.

3.12.2.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros L (linhas) e C (colunas).
2. **Parâmetro:** Ler o valor real $\alpha \in [0, 1]$.
3. **Dados:** Ler os valores inteiros da matriz f_1 (imagem 1) e em seguida da matriz f_2 (imagem 2).
4. **Mapeamento:** Para cada posição (i, j), calcular:

$$g(i,j) = \text{clip}\left(\text{round}\left(\alpha \cdot f_1(i,j) + (1-\alpha) \cdot f_2(i,j)\right)\right)$$

5. **Saída:** Exibir a matriz resultante $L \times C$.

3.12.2.2 📌 Restrições Computacionais

- **Arredondamento:** Aplicar `round` antes da conversão para inteiro.
- **Saturação:** Confinar ao intervalo $[0, 255]$ com $\text{clip}(x) = \max(0, \min(255, x))$.
- **Operação em float:** Realize a operação em ponto flutuante antes de arredondar.

3.12.2.3 🧠 Fundamentação Teórica

Valor de α	Resultado
$\alpha = 1.0$	Apenas f_1
$\alpha = 0.5$	Média aritmética de f_1 e f_2
$\alpha = 0.0$	Apenas f_2

3.12.2.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

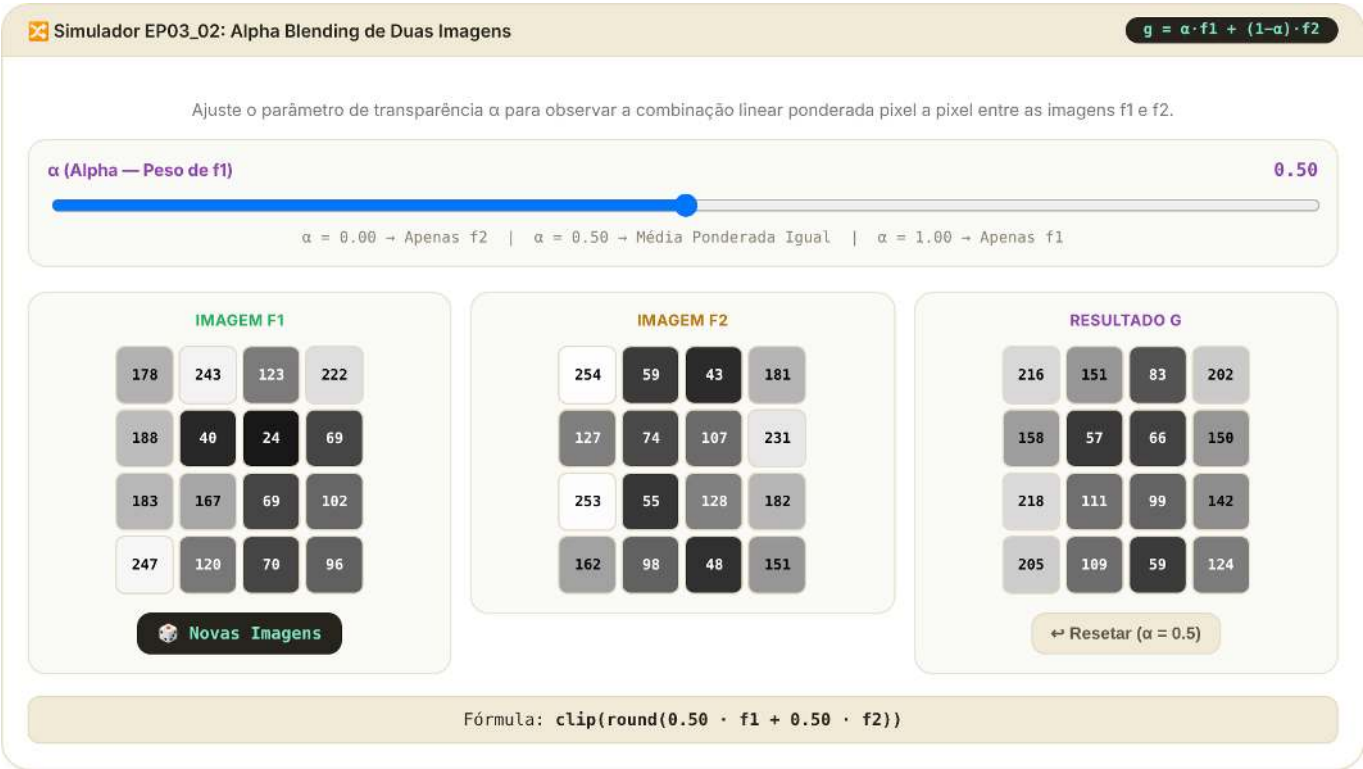
- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linha 3: Real α.
- Linhas seguintes: Elementos de f_1 (L linhas com C valores cada).
- Linhas seguintes: Elementos de f_2 (L linhas com C valores cada).

Saída:

- Matriz resultante $L \times C$.

3.12.2.5 📌 Exemplos

Entrada	Saída	Observação
1 3 0.5 0 100 200 100 200 50	50 150 125	Média entre as duas imagens
1 3 1.0 10 20 30 90 80 70	10 20 30	Apenas f_1 (alpha=1)



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap03/sim-ep0302-blending.html

Figura 3.27: Simulador EP03_02: Alpha Blending de Duas Imagens (g = · f1 + (1−) · f2)

```
%%writefile EP03_02.cpp
// sua solução
```

```
Overwriting EP03_02.cpp
```

```
TestSuite("EP03_02.cpp").run()
```

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

3.12.3 EP03_03 🎭 Inversão de Imagem (Negativo Fotográfico)

Em radiologia, as imagens de raio-X são tradicionalmente visualizadas em negativo: ossos aparecem em preto sobre fundo branco. A operação de **negativo fotográfico** é aplicada rotineiramente em PACS (*Picture Archiving and Communication Systems*) para facilitar a detecção de fraturas e densidades ósseas.

Ver na Figura 3.28 uma simulação deste EP.

3.12.3.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros L (linhas) e C (colunas).
2. **Dados:** Ler os valores inteiros da matriz original.
3. **Mapeamento:** Para cada pixel p, calcular o negativo:

$$p' = 255 - p$$

4. **Saída:** Exibir a matriz resultante $L \times C$.

3.12.3.2 📌 Restrições Computacionais

- **Sem *clipping* necessário:** O resultado de $255 - p$ com $p \in [0, 255]$ é sempre $\in [0, 255]$.
- **Tipo inteiro:** Saída deve ser valores inteiros.
- **Equivalência lógica:** A operação é idêntica ao `NOT` bit a bit (`mm::bnot`) em imagens de 8 bits.

3.12.3.3 🧠 Fundamentação Teórica

Pixel Original p	Pixel Negativo p'	Observação
0 (preto)	255 (branco)	Inversão total
128 (cinza médio)	127 (cinza médio)	Valor central
255 (branco)	0 (preto)	Inversão total

3.12.3.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linhas seguintes: Elementos inteiros da matriz original.

Saída:

- Matriz negativa em L linhas e C colunas.

3.12.3.5 📌 Exemplos

Entrada	Saída	Observação
140 128 200 255	255 127 55 0	Inversão de cada pixel
2 2 10 20 30 40	245 235225 215	Matriz 2x2 invertida

```
%%writefile EP03_03.cpp
// sua solução
```

```
Overwriting EP03_03.cpp
```

```
TestSuite("EP03_03.cpp").run()
```

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

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap03/sim-ep0303-inversao.html

Figura 3.28: Simulador EP03_03: Inversão de Imagem — Negativo Fotográfico (p' = 255 − p)

3.12.4 EP03_04 📊 Equalização de Histograma (L bits)

Em imagens de satélite de sensoriamento remoto, a variação de iluminação ao longo do dia produz imagens de baixo contraste. A **equalização de histograma** é aplicada automaticamente em satélites como o Landsat para redistribuir os tons, revelando detalhes de vegetação, relevo e zonas urbanas invisíveis na imagem original.

Ver na Figura 3.29 uma simulação deste EP.

3.12.4.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros L (linhas), C (colunas) e B (número de bits, com $L_{\max} = 2^B$).
2. **Dados:** Ler a matriz de pixels f com valores em $[0, 2^B - 1]$.
3. **Histograma:** Calcular $h[k]$ = número de pixels com intensidade k, para $k = 0 \ldots 2^B - 1$.
4. **Probabilidade:** $p[k] = h[k]/(L \cdot C)$.
5. **CDF:** $\text{cdf}[k] = \sum_{j=0}^{k} p[j]$; função de distribuição acumulada.
6. **LUT:** $\text{lut}[k] = \text{round}\,(\text{cdf}[k] \cdot (2^B - 1))$; *Look-Up Table* (tabela de consulta).
7. **Aplicação:** $g[i, j] = \text{lut}[f[i, j]]$.
8. **Saída:** Exibir a matriz equalizada $L \times C$.

3.12.4.2 📌 Restrições Computacionais

- **Arredondamento:** Usar arredondamento matemático (`round`) na LUT.
- **Bits:** O número de níveis é 2^B (ex.: $B = 3 \Rightarrow 8$ níveis, $B = 8 \Rightarrow 256$ níveis).
- **CDF acumulada:** $\text{cdf}[k] = \sum_{j=0}^{k} p[j]$, com $\text{cdf}[2^B - 1] = 1.0$.

3.12.4.3 🧠 Fundamentação Teórica

Etapa	Operação	Fórmula
1	Histograma	$h[k] \leftarrow$ nº pixels com intensidade k
2	Probabilidade	$p[k] = h[k]/(L \cdot C)$
3	CDF	$\text{cdf}[k] = \sum_{j=0}^{k} p[j]$

Etapa	Operação	Fórmula
4	LUT	$\text{lut}[Look-UpTable(tabeladeconsulta)k] = \text{round}(\text{cdf}[k] \cdot (2^B - 1))$
5	Aplicação	$g[i,j] = \text{lut}[f[i,j]]$

3.12.4.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linha 3: Inteiro B (número de bits).
- Linhas seguintes: Elementos inteiros da matriz.

Saída:

- Matriz equalizada em L linhas e C colunas.

3.12.4.5 📌 Exemplos

Entrada	Saída	Observação
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	Exemplo 3 bits do capítulo
1 4 3 0 0 7 7	0 0 7 7	Histograma bimodal extremo

```
%%writefile EP03_04.cpp
// sua solução
```

```
Overwriting EP03_04.cpp
```

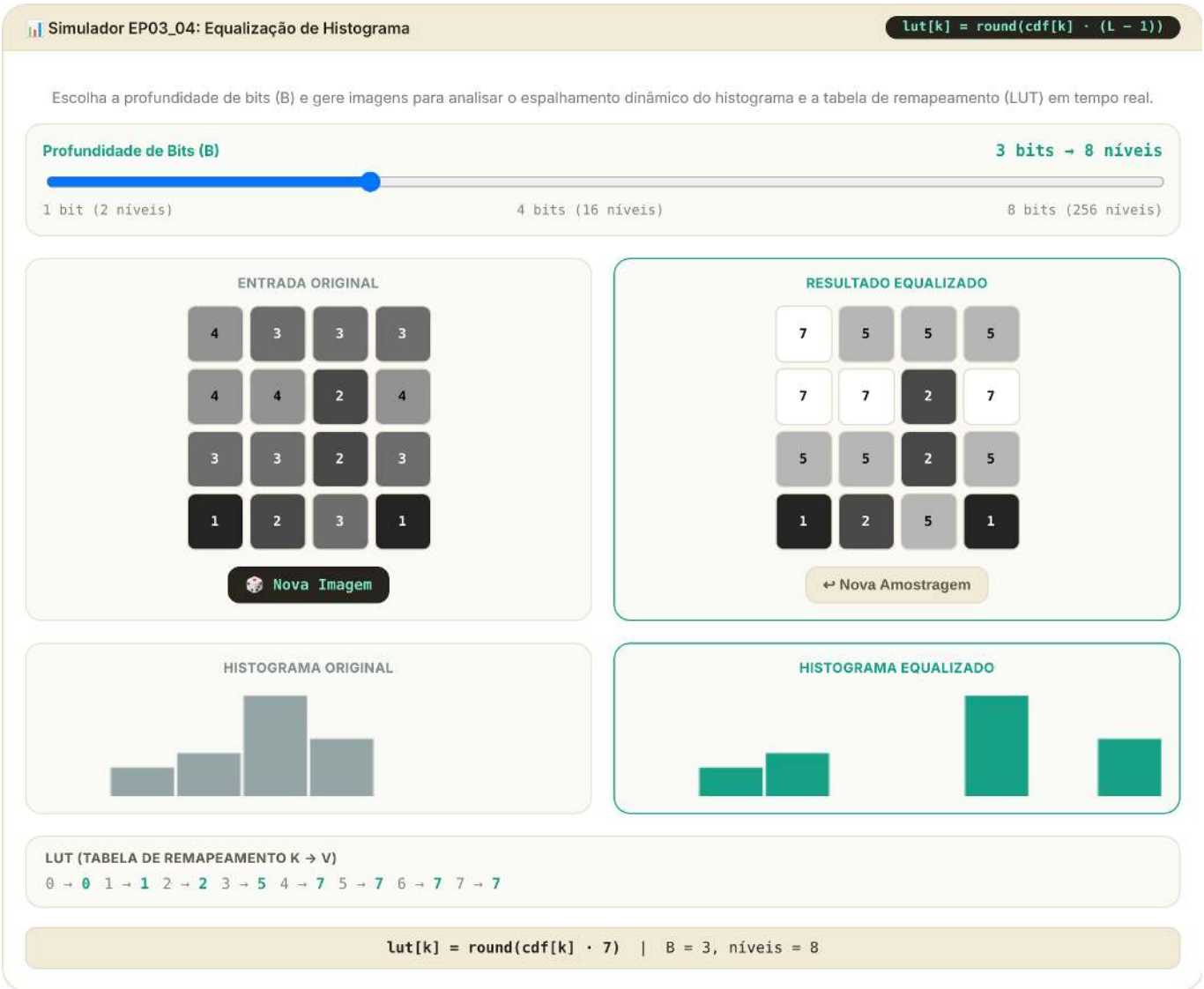
```
TestSuite("EP03_04.cpp").run()
```

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

3.12.5 EP03_05 🔲 Aplicação de Máscara AND Binária

Em sistemas de inspeção industrial por visão computacional, é necessário isolar regiões de interesse (ROI) em imagens de peças para verificar defeitos de fabricação. A operação **AND bit a bit com uma máscara binária** é o mecanismo fundamental para recortar exatamente a área de inspeção, zerando todos os pixels fora dela.

Ver na Figura 3.30 uma simulação deste EP.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap03/sim-ep0304-equalizacao.html

Figura 3.29: Simulador EP03_04: Equalização de Histograma (Níveis L = 2^B)

3.12.5.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros L (linhas) e C (colunas).
2. **Dados:** Ler a matriz de pixels f (valores $\in [0, 255]$).
3. **Máscara:** Ler a matriz binária m (valores: apenas 0 ou 255).
4. **Mapeamento:** Para cada pixel (i, j), aplicar o AND bit a bit:

$$g(i, j) = f(i, j) \text{ AND } m(i, j)$$

onde 255 = `11111111` e 0 = `00000000` em binário.

5. **Saída:** Exibir a matriz resultante $L \times C$.

3.12.5.2 📌 Restrições Computacionais

- **AND com 255:** p AND $255 = p$ (todos os bits preservados).
- **AND com 0:** p AND $0 = 0$ (todos os bits zerados).
- **Máscara:** Os únicos valores possíveis na máscara são 0 e 255.
- **Implementação:** Em Python, o AND bit a bit entre inteiros usa o operador `&`.

3.12.5.3 🧠 Fundamentação Teórica

Pixel f	Máscara m	Resultado f AND m
qualquer v	255 (`11111111`)	v (preservado)
qualquer v	0 (`00000000`)	0 (zerado)

3.12.5.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

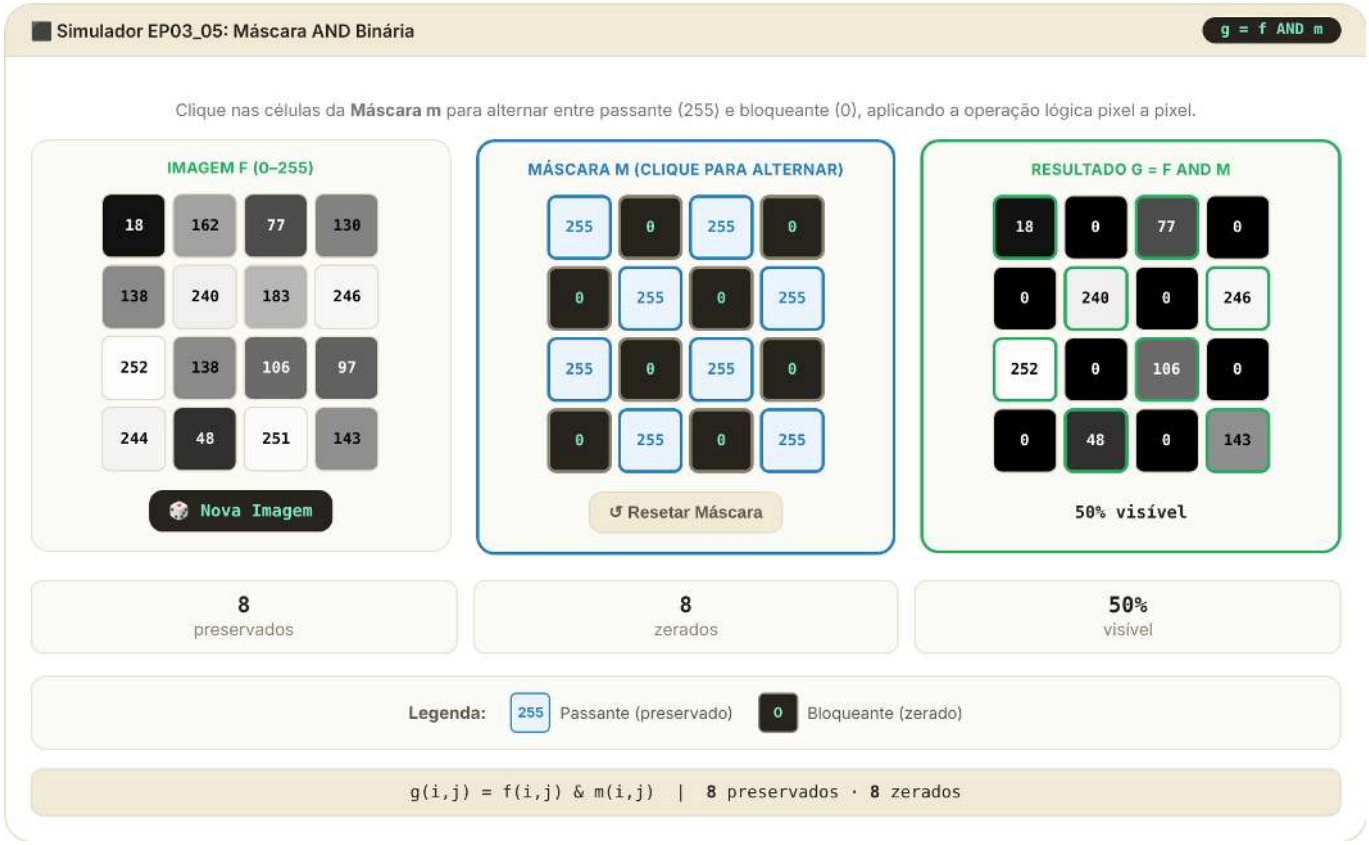
- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linhas seguintes: Elementos de f (L linhas).
- Linhas seguintes: Elementos de m (L linhas com valores 0 ou 255).

Saída:

- Matriz resultante $L \times C$.

3.12.5.5 📌 Exemplos

Entrada	Saída	Observação
2 3 100 150 200 50 80 120 255 255 0 0 255 255	100 150 0 0 80 120	Máscara seleciona região
1 4 10 20 30 40 255 0 255 0	10 0 30 0	Alternado preservado/zerado



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap03/sim-ep0305-mascara.html

Figura 3.30: Simulador EP03_05: Aplicação de Máscara AND Binária

```
%%writefile EP03_05.cpp
// sua solução
```

```
Overwriting EP03_05.cpp
```

```
TestSuite("EP03_05.cpp").run()
```

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

3.12.6 EP03_06 Filtro de Média com *Kernel* N×N

Em câmeras de veículos autônomos, imagens capturadas sob chuva ou névoa apresentam ruído gaussiano. O **filtro de média** é amplamente utilizado para sua redução em tempo real, sendo implementado diretamente no **ISP** (*Image Signal Processor*) de sensores **CMOS** (*Complementary Metal-Oxide-Semiconductor*).

Os sensores CMOS são os sensores de imagem usados na maioria das câmeras modernas (*smartphones*, *webcams*, câmeras automotivas etc.). Eles convertem a luz em sinais elétricos, e o ISP processa esses sinais em tempo real — aplicando operações como redução de ruído, balanço de branco e outros ajustes de imagem.

Ver na Figura 3.31 uma simulação deste EP.

3.12.6.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros L (linhas), C (colunas) e N (tamanho do *kernel*, sempre ímpar).
2. **Dados:** Ler a matriz de pixels f.
3. **Filtro de Média:** Para cada pixel (i, j) **interno** (sem bordas), calcular:

$$g(i,j) = \text{round}\left(\frac{1}{N^2}\sum_{s=-(r)}^{r}\sum_{t=-(r)}^{r} f(i+s, j+t)\right), \quad r = \lfloor N/2 \rfloor$$

4. **Tratamento de Borda:** Pixels na borda (onde a janela $N \times N$ ultrapassa os limites) devem ser **copiados diretamente** do original sem modificação.
5. **Saída:** Exibir a matriz resultante $L \times C$.

3.12.6.2 📌 Restrições Computacionais

- **Raio:** $r = \lfloor N/2 \rfloor$ (metade do *kernel*, inteiro).
- **Pixels internos:** (i, j) com $r \leq i < L - r$ e $r \leq j < C - r$.
- **Arredondamento:** Usar arredondamento matemático antes de converter para inteiro.
- **Sem *clipping*:** A média de valores $\in [0, 255]$ permanece em $[0, 255]$.

3.12.6.3 🧠 Fundamentação Teórica

Tamanho N	Coeficiente	Pixels na janela	Efeito
3	$1/9 \approx 0.111$	9	Suave
5	$1/25 = 0.04$	25	Médio
7	$1/49 \approx 0.020$	49	Forte

3.12.6.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

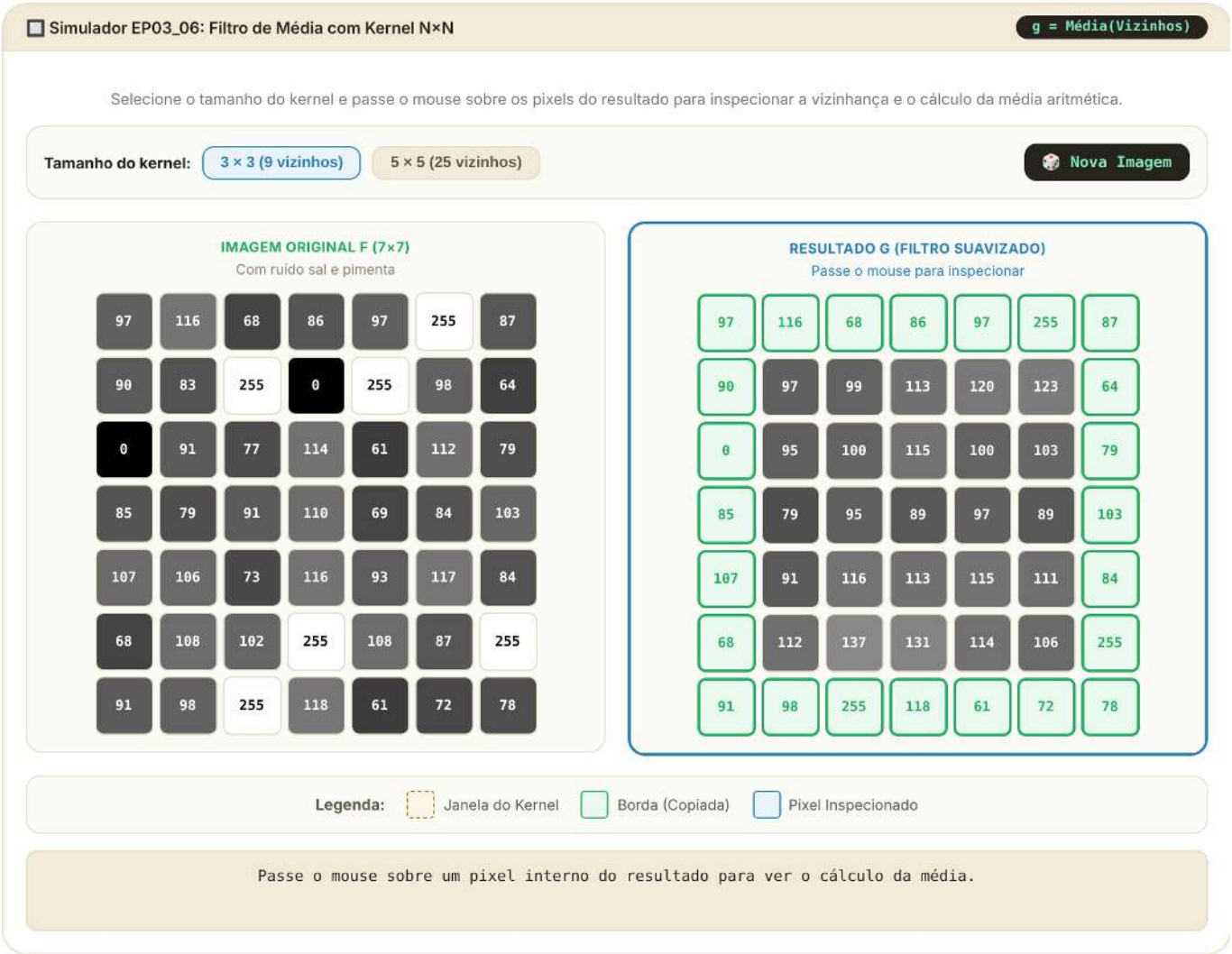
- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linha 3: Inteiro N (ímpar, $N \geq 3$).
- Linhas seguintes: Elementos da matriz original.

Saída:

- Matriz filtrada $L \times C$.

3.12.6.5 📌 Exemplos

Entrada	Saída	Observação
3 3 3 10 20 30 40 50 60 70 80 90	10 20 30 40 50 60 70 80 90	Apenas borda (3×3 = borda total)
5 5 3 0 0 0 0 0 0 0 0 0 0 0 0 100 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 11 11 11 0 0 11 11 11 0 0 11 11 11 0 0 0 0 0 0	Pixel isolado: todos os 9 pixels internos cuja janela 3×3 inclui o valor 100 recebem round(100/9)=11



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap03/sim-ep0306-media.html

Figura 3.31: Simulador EP03_06: Filtro de Média com Kernel N×N

```
%%writefile EP03_06.cpp
// sua solução
```

```
Overwriting EP03_06.cpp
```

```
TestSuite("EP03_06.cpp").run()
```

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

3.12.7 EP03_07 🔍 Operador Laplaciano (w4) para Realce de Bordas

Em tomografias de alta resolução, a nitidez das bordas entre tecidos é crítica para diagnóstico. O **operador Laplaciano** é amplamente utilizado em *pipelines* de pré-processamento de imagens médicas para realçar automaticamente os contornos anatômicos antes da segmentação, evitando intervenção manual do radiologista.

Ver na Figura 3.32 uma simulação deste EP.

3.12.7.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros L (linhas) e C (colunas).
2. **Dados:** Ler a matriz de pixels f.
3. **Laplaciano (w4):** Para cada pixel **interno** (i, j) com $1 \leq i < L-1$, $1 \leq j < C-1$, calcular:

$$\nabla^2 f(i,j) = f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1) - 4 \cdot f(i,j)$$

4. **Realce:** Calcular a imagem realçada:

$$g(i,j) = \text{clip}(f(i,j) - \nabla^2 f(i,j))$$

5. **Borda:** Pixels na borda são copiados diretamente: $g(i,j) = f(i,j)$.
6. **Saída:** Exibir a matriz realçada $L \times C$.

3.12.7.2 📌 Restrições Computacionais

- ***Kernel* w4:** $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ — apenas 4-vizinhos.
- **Saturação:** $\text{clip}(x) = \max(0, \min(255, x))$ aplicado ao resultado do realce.
- **Sem arredondamento:** O Laplaciano usa apenas somas/subtrações de inteiros.

3.12.7.3 🧠 Fundamentação Teórica

Região	$\nabla^2 f$	Efeito do Realce
Uniforme	≈ 0	Sem alteração
Borda crescente	< 0	Pixel clareado
Borda decrescente	> 0	Pixel escurecido

3.12.7.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linhas seguintes: Elementos da matriz original.

Saída:

- Matriz realçada $L \times C$.

3.12.7.5 📌 Exemplos

Entrada	Saída	Observação
3 3 0 0 0 0 100 0 0 0 0	0 0 0 0 255 0 0 0 0	Pico isolado: 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	Região uniforme: Laplaciano=0, sem alteração

```
%%writefile EP03_07.cpp
// sua solução
```

```
Overwriting EP03_07.cpp
```

```
TestSuite("EP03_07.cpp").run()
```

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

3.12.8 EP03_08 🧭 Gradiente de Sobel: Gx e Gy

Em robôs exploradores de Marte (como o Perseverance), a detecção de obstáculos é realizada em tempo real por câmeras estereoscópicas. O **operador de Sobel** calcula o gradiente direcional da cena e é utilizado no algoritmo de detecção de bordas para identificar rochas, fissuras e desníveis do terreno que possam comprometer a navegação.

Ver na Figura 3.33 uma simulação deste EP.

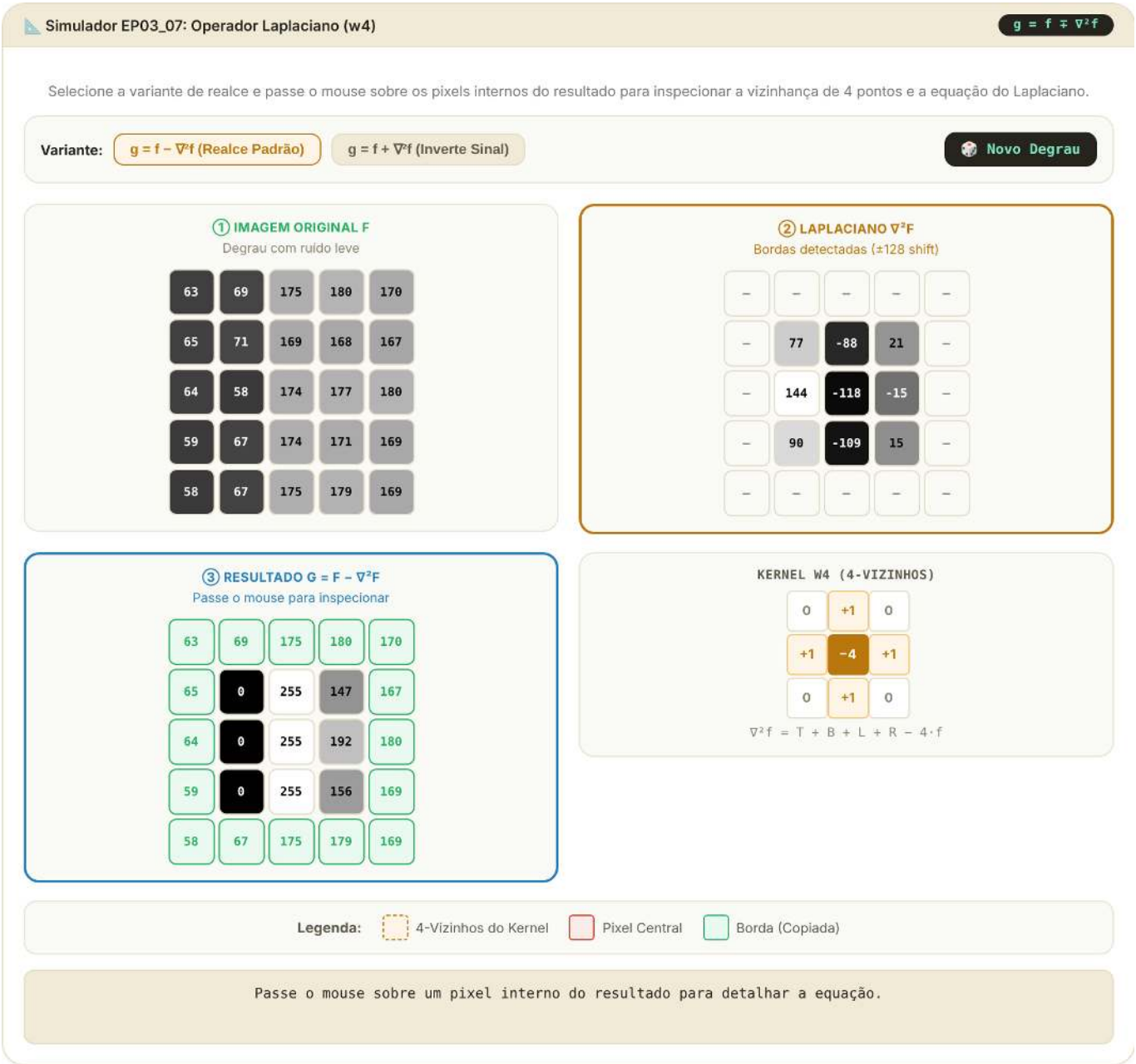
3.12.8.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros L (linhas) e C (colunas).
2. **Dados:** Ler a matriz f.
3. **Gx e Gy:** Para cada pixel **interno** (i, j) com $1 \leq i < L-1$, $1 \leq j < C-1$:

$$G_x(i,j) = [f(i-1,j+1) + 2f(i,j+1) + f(i+1,j+1)] - [f(i-1,j-1) + 2f(i,j-1) + f(i+1,j-1)]$$

$$G_y(i,j) = [f(i+1,j-1) + 2f(i+1,j) + f(i+1,j+1)] - [f(i-1,j-1) + 2f(i-1,j) + f(i-1,j+1)]$$

4. **Magnitude:** $|\nabla f(i,j)| = \text{clip}(\text{round}(\sqrt{G_x^2 + G_y^2}))$.
5. **Borda:** Pixels de borda recebem magnitude 0.
6. **Saída:** Exibir a magnitude $L \times C$.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap03/sim-ep0307-laplaciano.html

Figura 3.32: Simulador EP03_07: Operador Laplaciano (w4) para Realce de Bordas

3.12.8.2 📌 Restrições Computacionais

- **Arredondamento:** Aplicar `round` antes de converter para inteiro.
- **Saturação:** $\text{clip}(x) = \max(0, \min(255, x))$.
- **Raiz quadrada:** Usar $\sqrt{G_x^2 + G_y^2}$ (não a aproximação $|G_x| + |G_y|$).

3.12.8.3 🧠 Fundamentação Teórica

Operador	Detecta	Coeficientes diagonais
G_x	Bordas verticais	± 1
G_y	Bordas horizontais	± 1
$\lvert\nabla f\rvert$	Todas as bordas	Combinado

3.12.8.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linhas seguintes: Elementos da matriz.

Saída:

- Magnitude do gradiente, matriz $L \times C$.

3.12.8.5 📌 Exemplos

Entrada	Saída	Observação
3 3 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0	Imagem nula: gradiente zero
3 3 0 0 255 0 0 255 0 0 255	0 0 0 0 255 0 0 0 0	Borda vertical central: Gx alto

```
%%writefile EP03_08.cpp
// sua solução
```

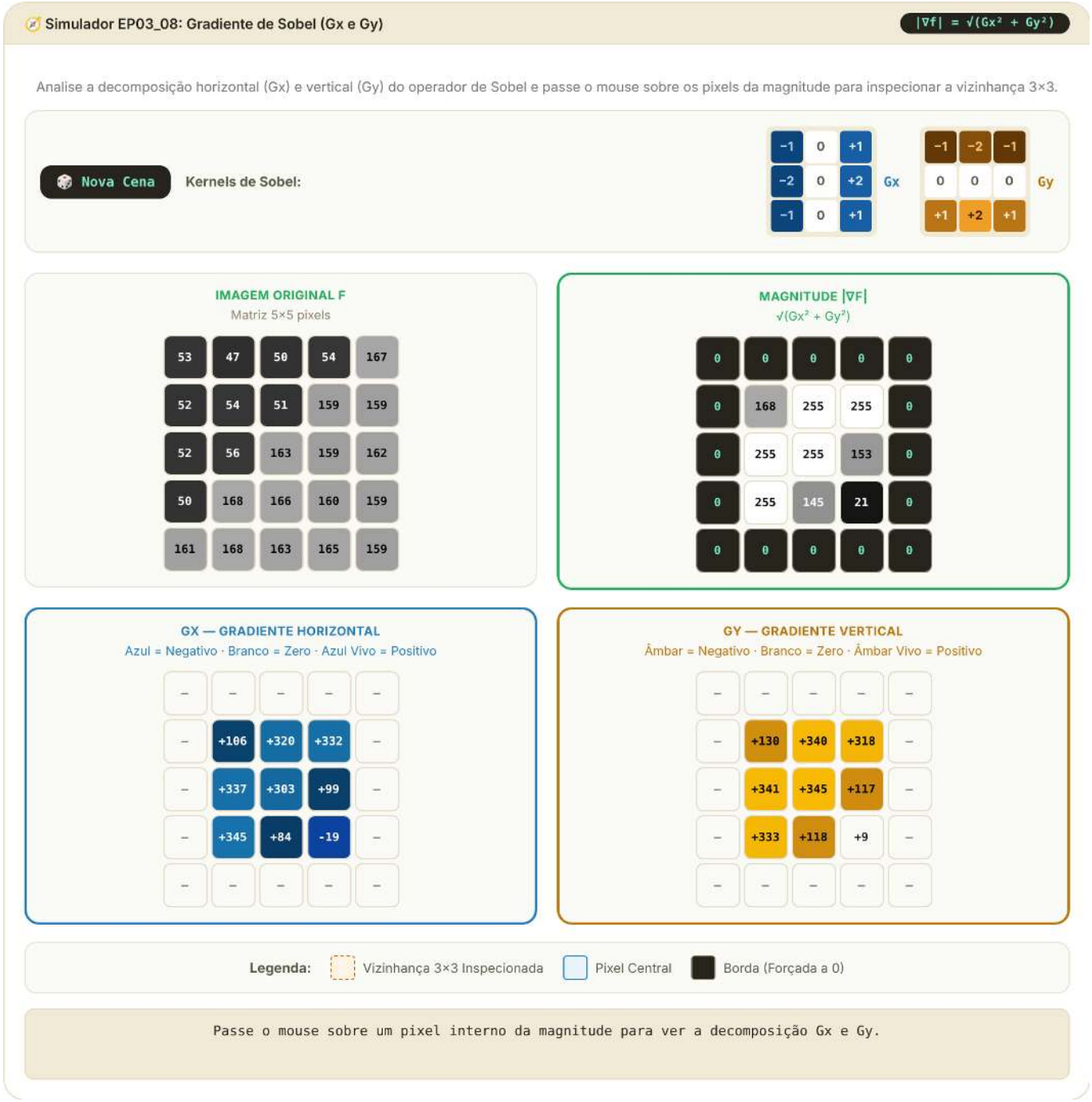
```
Overwriting EP03_08.cpp
```

```
TestSuite("EP03_08.cpp").run()
```

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

3.12.9 EP03_09 📡 Filtro da Mediana 3×3

Imagens de radar de abertura sintética (SAR) usadas em monitoramento ambiental e militar sofrem de um tipo específico de ruído chamado *speckle*, que possui características similares ao ruído sal e pimenta. O **filtro**



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap03/sim-ep0308-sobel.html

Figura 3.33: Simulador EP03_08: Gradiente de Sobel (Gx e Gy)

da mediana é o método padrão de remoção desse ruído porque preserva as bordas das estruturas enquanto elimina os pontos espúrios.

Ver na Figura 3.34 uma simulação deste EP.

3.12.9.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros L (linhas) e C (colunas).
2. **Dados:** Ler a matriz de pixels f.
3. **Filtro da Mediana 3×3:** Para cada pixel **interno** (i, j) com $1 \leq i < L-1$, $1 \leq j < C-1$:
   - Coletar os 9 pixels da vizinhança 3×3: $\{f(i+s, j+t) : s, t \in \{-1, 0, 1\}\}$.
   - Ordenar os 9 valores em ordem crescente.
   - Atribuir $g(i, j)$ ao valor central (posição índice 4, considerando índice 0).

$$g(i,j) = \text{mediana}\{f(i+s, j+t) : s, t \in \{-1, 0, 1\}\}$$

4. **Borda:** Copiar diretamente: $g(i, j) = f(i, j)$.
5. **Saída:** Exibir a matriz filtrada $L \times C$.

3.12.9.2 📌 Restrições Computacionais

- **Janela:** Sempre $3 \times 3 = 9$ elementos.
- **Mediana:** O elemento central da sequência ordenada (índice 4 de 0 a 8).
- **Sem clipping:** A mediana de valores em $[0, 255]$ permanece em $[0, 255]$.
- **Não linear:** O filtro mediana não pode ser expresso como convolução linear.

3.12.9.3 🧠 Fundamentação Teórica

Ruído	Filtro de Média	Filtro de Mediana
Sal e pimenta (0 ou 255)	Espalha o ruído	Remove sem distorcer bordas
Gaussiano	Reduz eficazmente	Reduz parcialmente

3.12.9.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linhas seguintes: Elementos da matriz.

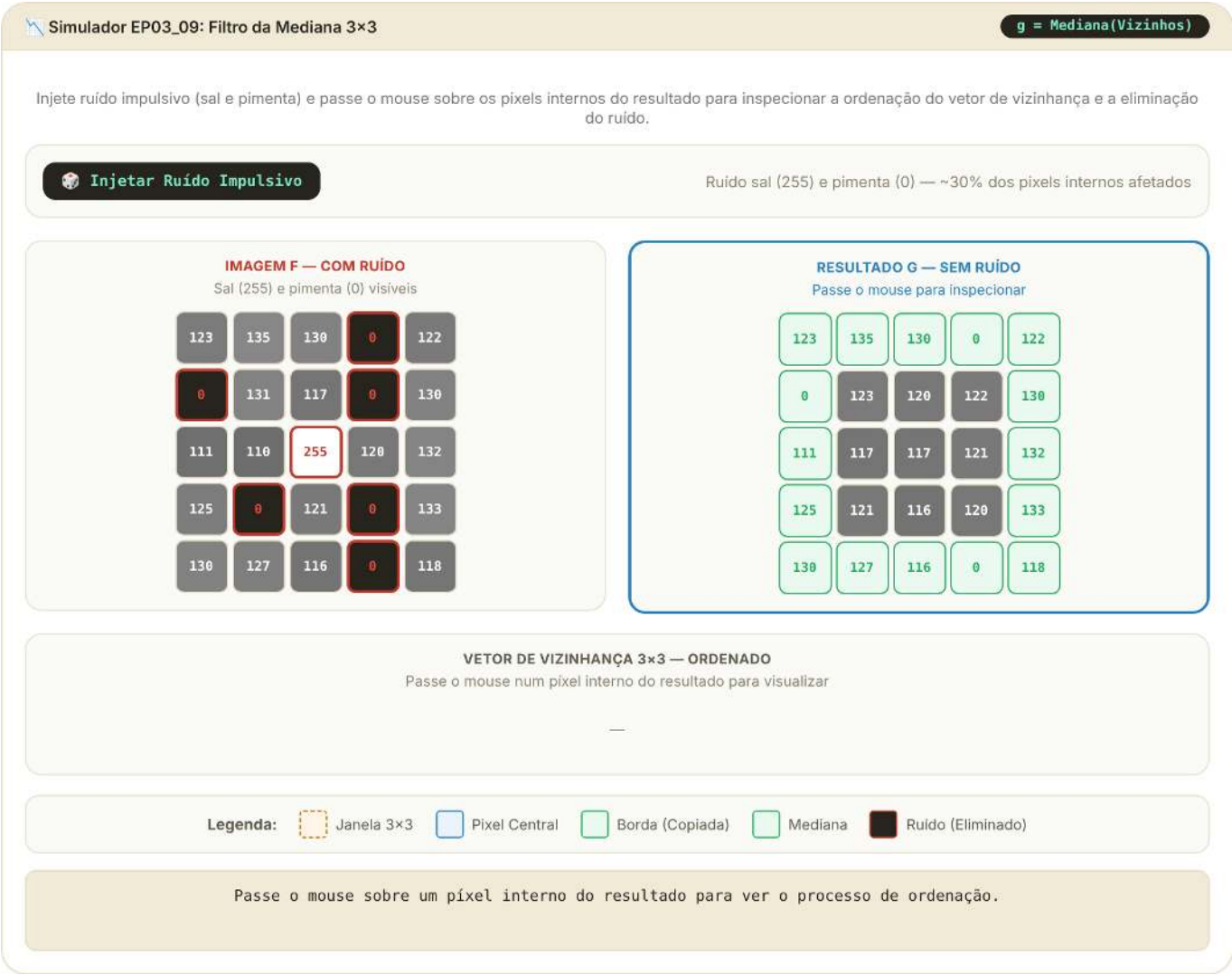
Saída:

- Matriz filtrada $L \times C$.

3.12.9.5 📌 Exemplos

Entrada	Saída	Observação
3 3 100 100 100 100 0 100 100 100 100	100 100 100 100 100 100 100 100 100	Ponto preto eliminado: mediana de $8\times100+1\times0 = 100$

Entrada	Saída	Observação
3 3 50 50 50 50 255 50 50 50 50	50 50 50 50 50 50 50 50 50	Ponto branco (sal) eliminado



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap03/sim-ep0309-mediana.html

Figura 3.34: Simulador EP03_09: Filtro da Mediana 3×3

```
%%writefile EP03_09.cpp
// sua solução
```

```
Overwriting EP03_09.cpp
```

```
TestSuite("EP03_09.cpp").run()
```

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

3.12.10 EP03_10 ✨ *Unsharp Masking* (USM)

Em sistemas de digitalização de documentos históricos e obras de arte, a nitidez das imagens é fundamental para leitura de textos manuscritos e detalhes ornamentais. O ***Unsharp Masking*** **(USM)** é o algoritmo de realce de nitidez padrão utilizado em *scanners* profissionais e softwares como Adobe Photoshop, controlado pelo parâmetro k que determina a intensidade do realce.

Ver na Figura 3.35 uma simulação deste EP.

3.12.10.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros L (linhas) e C (colunas).
2. **Parâmetro:** Ler o valor real k (intensidade do realce, $k \geq 0$).
3. **Dados:** Ler a matriz de pixels f.
4. **Suavização:** Calcular $\bar{f}$ com filtro de média 3×3 (apenas pixels internos; bordas mantidas):

$$\bar{f}(i,j) = \frac{1}{9} \sum_{s=-1}^{1} \sum_{t=-1}^{1} f(i+s, j+t)$$

5. **Máscara de alta frequência:** $m(i,j) = f(i,j) - \bar{f}(i,j)$.
6. **Realce USM:** Para cada pixel interno:

$$g(i,j) = \text{clip}\left(\text{round}\left(f(i,j) + k \cdot m(i,j)\right)\right)$$

7. **Borda:** $g(i,j) = f(i,j)$ (cópia direta).
8. **Saída:** Exibir a matriz realçada $L \times C$.

3.12.10.2 📌 Restrições Computacionais

- **Arredondamento:** Aplicar `round` antes do clipping.
- **Saturação:** $\text{clip}(x) = \max(0, \min(255, x))$.
- **Operações em float:** Calcular $\bar{f}$ e m em ponto flutuante antes de arredondar o resultado final.
- $k = 0$**:** Sem realce — a saída é idêntica à entrada (exceto pelas bordas).

3.12.10.3 🧠 Fundamentação Teórica

Etapa	Operação	Descrição
1	$\bar{f} = f * \frac{1}{9}\mathbf{1}_{3\times3}$	Suavização (baixas frequências)
2	$m = f - \bar{f}$	Máscara (altas frequências)
3	$g = \text{clip}(\text{round}(f + k \cdot m))$	Realce ponderado

3.12.10.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linha 3: Real k.
- Linhas seguintes: Elementos da matriz original.

Saída:

- Matriz realçada $L \times C$.

3.12.10.5 📌 Exemplos

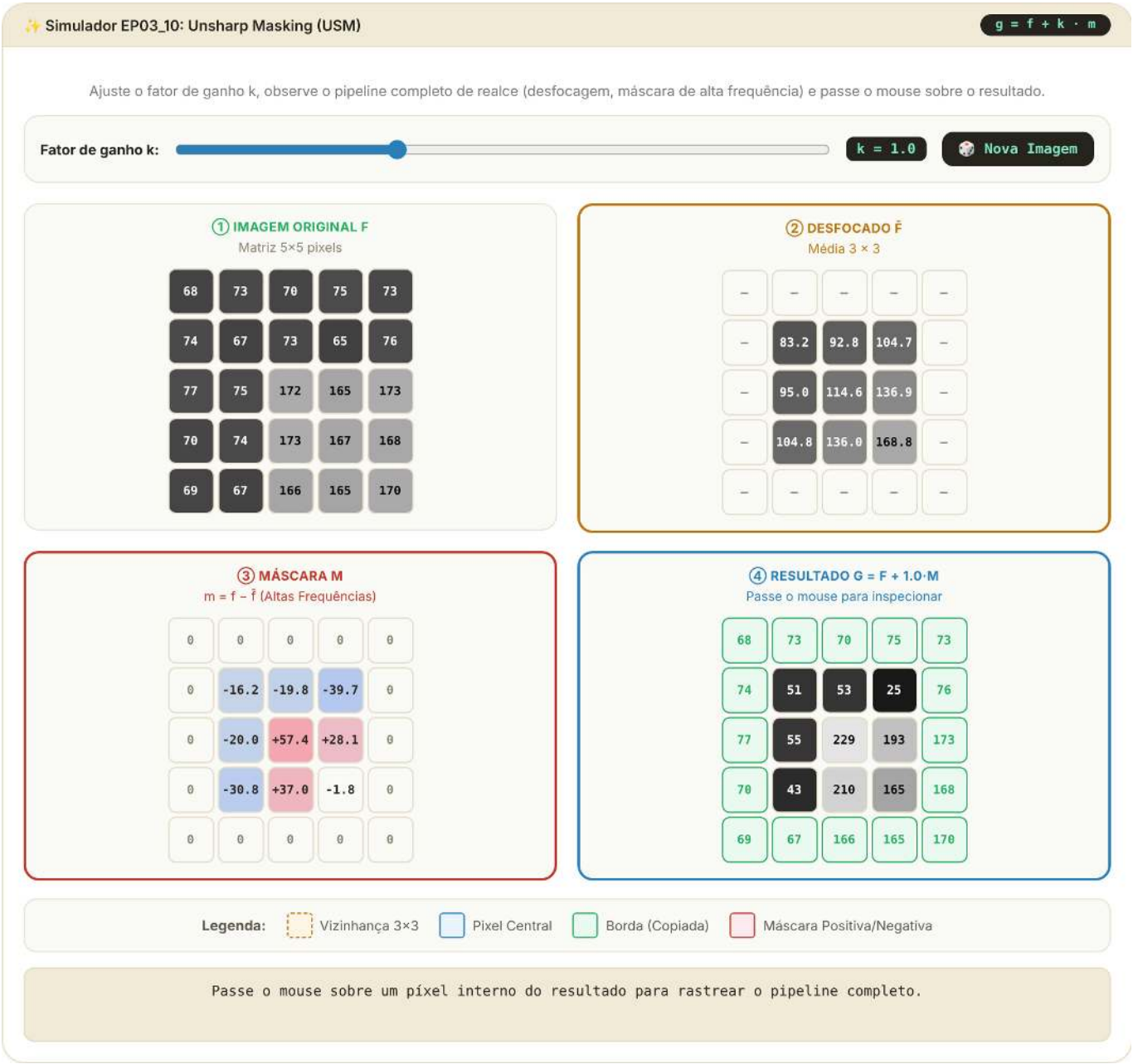
Entrada	Saída	Observação
3 3 0.0 100 100 100 100 100 100 100 100 100	100 100 100 100 100 100 100 100 100	k=0: sem realce
3 3 1.0 50 50 50 50 200 50 50 50 50	50 50 50 50 255 50 50 50 50	k=1: pixel central realçado e saturado

```
%%writefile EP03_10.cpp
// sua solução
```

```
Overwriting EP03_10.cpp
```

```
TestSuite("EP03_10.cpp").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap03/sim-ep0310-unsharp.html

Figura 3.35: Simulador EP03_10: *Unsharp Masking* (USM)

Capítulo 4

Morfologia Matemática e Segmentação de Imagens

Este capítulo apresenta dois temas fundamentais do Processamento Digital de Imagens (PDI): a **morfologia matemática** e a **segmentação de imagens**. A morfologia matemática fornece um arcabouço teórico baseado na teoria dos conjuntos para analisar, refinar e quantificar a forma de objetos em imagens binárias e em tons de cinza, por meio de operadores fundamentais como **erosão** e **dilatação**. A segmentação, por sua vez, tem como objetivo particionar a imagem em regiões de interesse, separando objetos do fundo e produzindo representações adequadas para análise e interpretação.

O capítulo inicia com a **limiarização**, uma das técnicas mais importantes de segmentação, introduzindo o método automático de **Otsu** e revisitando a análise de histogramas por meio da variância interclasses, apresentada no Capítulo 1. Em seguida, são estudados os principais operadores da morfologia matemática, incluindo erosão, dilatação, abertura, fechamento e reconstrução morfológica, que permitem refinar máscaras binárias e preservar estruturas relevantes dos objetos. Por fim, são apresentadas técnicas de segmentação baseada em regiões, como a **rotulação de componentes conexos**, a **transformada de distância** e o algoritmo ***watershed*** baseado em marcadores, culminando na extração de descritores geométricos e na geração de *bounding boxes* compatíveis com sistemas modernos de detecção de objetos.

4.1 Objetivos

Ao final deste capítulo, você será capaz de:

- **Aplicar limiarização:** Compreender o critério automático de Otsu por maximização da variância interclasses (σ_B^2) e selecionar estratégias adequadas de pré-processamento para facilitar a segmentação;
- **Dominar morfologia binária:** Compreender e aplicar erosão ($A \ominus B$) e dilatação ($A \oplus B$) como operadores fundamentais, derivando abertura ($A \circ B$), fechamento ($A \bullet B$) e operações baseadas em reconstrução morfológica, como `mm::clohole` e `mm::edgeoff`;
- **Aplicar morfologia em tons de cinza:** Utilizar gradiente morfológico e filtros *top-hat* para realce e análise de estruturas locais;
- **Rotular componentes conexos:** Identificar e separar regiões conectadas em imagens binárias por meio de algoritmos de rotulação;
- **Aplicar transformada de distância:** Interpretar e calcular distâncias ao fundo utilizando abordagens morfológicas e métricas geométricas;
- **Segmentar por regiões:** Construir *pipelines* de segmentação baseados em marcadores utilizando Transformada de Distância e o algoritmo ***watershed***;
- **Extrair descritores geométricos:** Calcular propriedades como área, perímetro, centróide, circularidade e *bounding boxes* por meio de `mm::label0` e extração de contornos;
- **Relacionar PDI e visão computacional:** Compreender como descritores extraídos por segmentação podem ser convertidos para formatos utilizados por detectores modernos, como YOLO.

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # artefatos de build da trilha C++ (.cpp, binário,
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")

# O kernel é Python mesmo na trilha C++: `mm` (morph.py) é usado pelos
# simuladores, pela exibição das figuras que o binário C++ gera e pelo
# estado mm::Image entre células. cpp=True baixa também a trilha compilada
# (morph.hpp + stb_image*.h), usada no #include das células %%writefile *.cpp.
# As células C++ deste capítulo compilam COM OpenCV (-DMM_USE_OPENCV +
# pkg-config opencv4): mm::dil/ero → cv::dilate/erode, então open/close/asf/
# gradm/tophat/blackhat rodam full-res e batem bit a bit com a trilha py.
import config
config.setup(cpp=True)
from morph import mm
import numpy as np
```

```
Ambiente pronto. Morph: 1.1.9 | OpenCV: 5.0.0
```

4.2 Limiarização

A **limiarização** (*thresholding*) é uma das formas mais simples e eficientes de segmentação de imagens. Seu objetivo é classificar cada pixel em duas classes de intensidade, normalmente associadas a *objeto* e *fundo*:

$$g(x,y) = \begin{cases} 255, & \text{se } f(x,y) > T \\ 0, & \text{caso contrário} \end{cases} \tag{4.1}$$

em que $f(x,y)$ representa a intensidade do pixel na imagem original e $g(x,y)$ a imagem binária resultante.

A escolha do limiar T é importante para a qualidade da segmentação. O método de **Otsu** determina automaticamente o limiar ótimo ao maximizar a **variância interclasses** σ_B^2 e definida por:

$$\sigma_B^2(T) = w_0(T)\, w_1(T)\, \left[\mu_0(T) - \mu_1(T)\right]^2 \tag{4.2}$$

em que:

- $w_0(T)$ e $w_1(T)$ são as probabilidades acumuladas das classes fundo e objeto;
- $\mu_0(T)$ e $\mu_1(T)$ são as médias de intensidade dessas classes;
- $\sigma_B^2(T)$ representa a variância interclasses para um dado limiar T.

O método funciona melhor quando o histograma apresenta dois grupos de intensidades relativamente separados. Para isso, o algoritmo avalia todos os limiares possíveis da imagem — tipicamente no intervalo $[0, 255]$ para imagens de 8 bits — e seleciona o valor que maximiza a variância entre classes, denotada por σ_B^2:

$$T^* = \arg\max_{T \in [0,255]} \sigma_B^2(T)$$

i Otsu assume histogramas bimodais

O método de Otsu produz melhores resultados quando o histograma apresenta dois picos bem definidos (*bimodalidade*), correspondentes ao fundo e ao objeto. Quanto maior a separação entre esses picos e mais pronunciado o máximo de σ_B^2, mais confiável tende a ser o limiar obtido.
Em imagens com iluminação não uniforme ou múltiplas regiões de intensidade, técnicas de **limiarização adaptativa** — nas quais o limiar é calculado localmente — costumam produzir segmentações mais robustas.
O índice B em σ_B^2 significa ***between classes*** (*entre classes*). Assim, σ_B^2 representa a **variância entre as classes** (*between-class variance*).

4.2.1 Imagem de Moedas

A imagem utilizada para praticar a segmentação é uma fotografia de coleção de moedas de diferentes países e épocas (Figura 4.1). Crédito: GAZI.MD.AHAD (CC BY-SA 4.0). Ela apresenta objetos circulares com bordas bem definidas, sendo ideal para demonstrar limiarização, operadores morfológicos, transformada de distância, *watershed* e descritores de forma.

```
%%writefile tmp/fig_04_coins.cpp
#define MM_OUT "tmp/fig_04_coins.png"
//| label: fig-04-coins
//| fig-cap: "Imagem com moedas de vários tipos. Crédito: GAZI.MD.AHAD (CC BY-SA 4.0)."
//| echo: true

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

int main() {
    // A imagem já está no diretório do capítulo (imagens/coins.png); a trilha
    // Python cuida do download/cache quando o notebook roda avulso.
    mm::Image img_coins_color = mm::read("imagens/coins.png");
    mm::Image img_coins_gray  = mm::gray(img_coins_color);

    std::cout << "Dimensões [y,x,c]: [" << img_coins_color.h << ", "
              << img_coins_color.w << ", " << img_coins_color.channels << "]" << std::endl;
    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"
```

```
Dimensões [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)")
```

Figura 4.1: Imagem com moedas de vários tipos. Crédito: GAZI.MD.AHAD (CC BY-SA 4.0).

4.2.2 Pré-processamento para o Otsu

O método de Otsu depende de um histograma bem **bimodal**. A imagem das moedas tem iluminação não uniforme e moedas escuras próximas do fundo, o que atrapalha. Na trilha C++ aplica-se **equalização global de histograma** (`mm::equalize`) antes da limiarização — a comparação CLAHE × Gaussiano e as curvas $\sigma_B^2(T)$ ficam na trilha Python (Figura 4.2).

```
%%writefile tmp/fig_04_otsu_comparacao_histogramas.cpp
#define MM_OUT "tmp/fig_04_otsu_comparacao_histogramas.png"
```

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

//| label: fig-04-otsu-comparacao-histogramas
//| fig-cap: "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
$\\sigma^2_B(T)$.)"
//| echo: true
//| output: true

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]

    mm::Image img_clahe = mm::clahe(img_coins_gray, 2.0, 8);   // realce adaptativo com limite
de contraste
    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)")
```

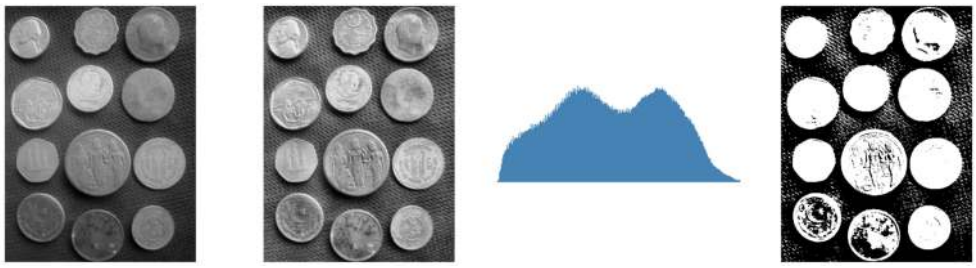


Figura 4.2: CLAHE (realce adaptativo com limite de contraste) antes da limiarização de Otsu: original, realçado, histograma e binarização. (A trilha Python também traça as curvas de $\sigma_B^2(T)$.)

4.2.3 Resultado: CLAHE como Melhor Pré-processamento

A análise da Figura 4.2 indica que o **CLAHE** obteve o maior valor da variância inter-classes ($\sigma_B^2 \approx 2{,}47 \times 10^3$), com limiar ótimo $T^* = 122$. Embora a combinação **CLAHE+Gaussiano** tenha produzido resultado muito semelhante ($\sigma_B^2 \approx 2{,}43 \times 10^3$, $T^* = 123$), o critério quantitativo do método de Otsu favorece ligeiramente o uso do CLAHE isoladamente.

Em termos visuais, as imagens binarizadas obtidas com CLAHE e CLAHE+Gaussiano são praticamente equivalentes. A diferença entre as duas abordagens torna-se mais evidente na análise dos histogramas e dos valores de $\sigma_B^2(T)$ do que na inspeção direta das segmentações resultantes. Assim, a escolha do CLAHE baseia-se principalmente na maximização da separação estatística entre as classes de fundo e objeto.

> Interpretação dos resultados
>
> Observe que os pré-processamentos com CLAHE e CLAHE+Gaussiano produzem histogramas e limiares ótimos muito próximos ($T^* = 122$ e $T^* = 123$). Consequentemente, as imagens binarizadas resultantes também são bastante semelhantes. Nesse caso, a decisão não é baseada em diferenças visuais marcantes, mas no critério objetivo do método de Otsu: o maior valor de σ_B^2 indica a melhor separação entre as classes.

4.3 Morfologia Matemática

A **morfologia matemática** é uma teoria baseada em conjuntos utilizada para analisar a forma e a estrutura de objetos em imagens. Diferentemente dos filtros lineares apresentados no Capítulo 3, os operadores morfológicos são **não lineares**, pois se baseiam em operações de mínimo, máximo e inclusão espacial, em vez de combinações lineares de intensidades. Esses operadores atuam sobre a vizinhança de cada pixel por meio de um **elemento estruturante** $\mathbb{B}$, responsável por definir a forma e o tamanho da região analisada.

Em imagens binárias e em tons de cinza com elementos planos, o elemento estruturante transladado para a posição x é definido espacialmente como:

$$\mathbb{B}_x = \{x + b \mid b \in \mathbb{B}\}$$

Nas regiões de borda da imagem, parte do conjunto $\mathbb{B}_x$ pode extrapolar o domínio físico da cena ($\mathbb{E}$). Para garantir a consistência matemática dos operadores primitivos nessas fronteiras, assume-se teoricamente que o espaço exterior ao domínio da imagem é preenchido com o **elemento neutro** da operação correspondente (infinito positivo para a erosão e infinito negativo para a dilatação), impedindo que o ambiente externo corrompa as estruturas internas do objeto.

Quando o elemento estruturante associa pesos a seus elementos — isto é, $b : \mathbb{B} \to \mathbb{Z}$ — ele é denominado **função estruturante** ou **elemento estruturante não plano**.

Desenvolvida por Matheron e Serra na década de 1960 para imagens binárias e posteriormente estendida a tons de cinza, a morfologia matemática fundamenta operadores como gradiente morfológico, *top-hat*, *watershed* e transformada de distância, todos derivados de dois primitivos: **erosão** e **dilatação** [Matheron (1975); Serra (1982)].

4.3.1 Erosão e Dilatação

Os dois operadores primitivos são definidos de forma unificada para imagens em tons de cinza ($f : \mathbb{E} \to \mathbb{Z}$) e, por restrição ao domínio $\{0, 1\}$, também para imagens binárias.

4.3.1.1 Erosão

A **Erosão** de uma imagem f por uma função estruturante $b : \mathbb{B} \to \mathbb{Z}$ é definida formalmente por:

$$\varepsilon_b(f)(x) = (f \ominus b)(x) = \min_{z \in \mathbb{B}} \{\, f(x+z) - b(z) \,\}, \quad \forall\, x \in \mathbb{E} \tag{4.3}$$

Na prática, a erosão substitui a intensidade do pixel x pelo mínimo valor resultante da diferença entre a imagem e o elemento estruturante na vizinhança definida pelo domínio $\mathbb{B}$. Valores positivos nos pesos de $b(z)$ forçam o resultado local para baixo, "cavando" o relevo da imagem mais profundamente e intensificando a erosão.

No caso **plano** (onde os pesos são nulos dentro do domínio, ou seja, $b \equiv 0$), a expressão simplifica-se para o mínimo local puro:

$$\varepsilon_B(f)(x) = \min\{\, f(y) : y \in \mathbb{B}_x \,\}$$

Em imagens binárias, essa operação equivale a exigir que o conjunto $\mathbb{B}$, transladado para a coordenada x, esteja **completamente contido** no objeto A:

$$A \ominus \mathbb{B} = \{\, z \in \mathbb{E} \mid \mathbb{B}_z \subseteq A \,\}$$

Efeito Visual: *Encolhe* objetos e estruturas claras, eliminando protuberâncias, picos brilhantes ou ruídos que sejam geometricamente menores que o domínio $\mathbb{B}$.

4.3.1.2 Implementação da erosão

A versão didática `mm::ero0` implementa o caso particular de erosão com **elemento estruturante plano**. Para cada pixel (y, x), a função percorre os vizinhos espaciais permitidos por B e armazena o menor valor encontrado na imagem de entrada f, reproduzindo diretamente a operação de mínimo local descrita na Equação 4.3 para $b \equiv 0$.

Observe que os valores dos vizinhos são sempre lidos de forma estática da imagem original f; a matriz de saída g é utilizada exclusivamente para registrar o mínimo acumulado da vizinhança corrente. Dessa forma, o resultado final é invariante em relação à ordem de varredura dos pixels (seja por linhas ou colunas).

A função auxiliar `_viz` calcula as coordenadas dos vizinhos válidos dentro dos limites físicos da imagem. Nas bordas, a inicialização do acumulador em `255` emula com exatidão o preenchimento por elemento neutro exigido pela teoria. Já a função de interface `mm::ero` recorre à implementação nativa e otimizada do OpenCV (`mm::ero`) quando o elemento estruturante é plano, chaveando para a rotina geral `mm::ero1` caso o elemento possua pesos topográficos.

O exemplo computacional a seguir ilustra a aplicação de um elemento estruturante em cruz (`mm::secross()`) em destaque na Figura 4.3, comparando a execução da variante didática em laço (`mm::ero0`) com o motor computacional do 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"
#include <iostream>

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)")
```

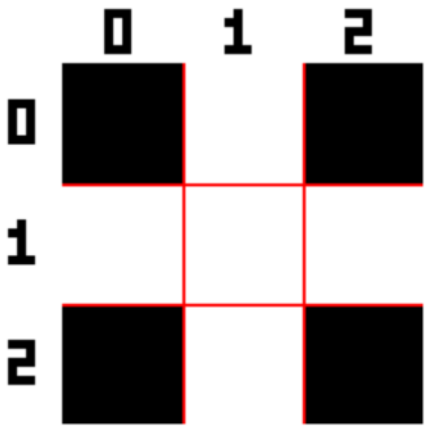


Figura 4.3: Elemento estruturante em formato de cruz (B_{cruz}) utilizado para conectividade-4.

```
# A morph.hpp é header-only; abaixo, o helper _viz e o corpo de 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 Dilatação

A **Dilatação** de uma imagem f por uma função estruturante $b : \mathbb{B} \to \mathbb{Z}$ é definida formalmente por:

$$\delta_b(f)(x) = (f \oplus b)(x) = \max_{z \in \mathbb{B}} \{ f(x - z) + b(z) \}, \quad \forall\, x \in \mathbb{E} \tag{4.4}$$

Na prática, a dilatação substitui a intensidade do pixel x pelo maior valor resultante da soma entre a imagem e o elemento estruturante na vizinhança definida. O argumento de inversão espacial $(x - z)$ indica que a dilatação avalia implicitamente o elemento transposto (refletido) $\hat{b}$, propriedade fundamental para assegurar a dualidade matemática em relação à erosão.

No caso **plano** (onde os pesos são nulos dentro do domínio, ou seja, $b \equiv 0$), a expressão reduz-se ao máximo local puro:

$$\delta_B(f)(x) = \max\{ f(y) : y \in \mathbb{B}_x \}$$

Em imagens binárias, essa operação equivale a exigir que o conjunto refletido $\hat{\mathbb{B}}$, transladado para a coordenada x, possua **interseção não vazia** com o objeto A:

$$A \oplus \mathbb{B} = \{ z \in \mathbb{E} \mid \hat{\mathbb{B}}_z \cap A \neq \emptyset \}$$

Efeito Visual: *Expande* as estruturas claras da imagem, aumentando o preenchimento de objetos, conectando componentes próximos e eliminando canais, fossas escuras ou vales que sejam geometricamente menores que o domínio $\mathbb{B}$.

4.3.1.4 Implementação da dilatação

A versão didática `mm::dil0` implementa o caso particular de dilatação com **elemento estruturante plano**. Para cada pixel (y, x), a função percorre os vizinhos espaciais permitidos por B e armazena o maior valor encontrado na imagem de entrada f, reproduzindo diretamente a operação de máximo local para $b \equiv 0$.

Antes de iniciar a varredura espacial, o elemento estruturante sofre uma reflexão geométrica por meio da instrução `np.flip(Bc)` para construir explicitamente a matriz transposta $\hat{B}$ exigida pela teoria. Em máscaras perfeitamente simétricas (como cruzes, quadrados e discos centrados na origem), essa reflexão não altera o arranjo dos pixels; contudo, para elementos assimétricos, tal etapa é estritamente necessária para garantir a equivalência com as definições formais e salvaguardar as leis de dualidade.

Assim como verificado no operador de erosão, os valores dos vizinhos são sempre lidos de forma estática a partir da matriz original f, enquanto a matriz de saída g atua puramente como o registrador do máximo acumulado da vizinhança. Nas fronteiras da imagem, a inicialização do acumulador em `0` emula com exatidão o preenchimento externo por elemento neutro ($-\infty$, ou zero em representações de 8 bits), garantindo que as bordas físicas da cena sejam dilatadas em perfeita conformidade com o padrão adotado pelo OpenCV.

O exemplo computacional abaixo ilustra a aplicação prática de um elemento em cruz (`mm::secross()`), validando a consistência entre a lógica em laços (`mm::dil0`) e o método nativo industrial (`mm::dil`).

```
# Corpo de mm::dil0() na morph.hpp (reflete o SE antes de varrer, como 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 Nota: O Confronto dos Sinais ($f(x+z)$ vs $f(x-z)$)

Compare as definições formais da erosão (Equação 4.3) e da dilatação (Equação 4.4). Considere um elemento estruturante assimétrico à direita $\mathbb{B} = \{0, 1\}$ (origem e um pixel à direita) aplicado na posição $x = 10$.

1. **Na Erosão** (Equação 4.3):

$$\min\{f(x+z) - b(z)\}$$

- $z = 0 \Rightarrow f(10+0) = \mathbf{f(10)}$
- $z = 1 \Rightarrow f(10+1) = \mathbf{f(11)}$

O operador consulta o pixel atual (10) e o pixel à direita (11), preservando a orientação original

de $\mathbb{B}$.

2. **Na Dilatação** (Equação 4.4):

$$\max\{f(x-z)+b(z)\}$$

   - $z=0 \Rightarrow f(10-0)=\mathbf{f(10)}$
   - $z=1 \Rightarrow f(10-1)=\mathbf{f(9)}$

   Devido ao sinal negativo $(-z)$, avançar no elemento estruturante corresponde a recuar na imagem, fazendo com que a dilatação consulte o pixel à esquerda (9).

A função `_viz`, utilizada em `morph.py`, gera vizinhos por deslocamentos aditivos da forma $x+z$. Por esse motivo, a implementação de `mm::dil0` reflete previamente o elemento estruturante por meio de `np.flip(B)`. Após a reflexão, a varredura baseada em $x+z$ passa a acessar exatamente os mesmos pontos definidos pela expressão teórica $f(x-z)$ da dilatação em Equação 4.4.
Para elementos estruturantes simétricos (como discos, quadrados e cruzes centradas), a reflexão não altera a máscara. Já para elementos assimétricos, essa etapa é indispensável para que a implementação reproduza corretamente a definição matemática da dilatação e preserve a dualidade erosão–dilatação.

i Dualidade erosão–dilatação

Erosão e dilatação são **duais pelo complemento**. Isso significa que um operador pode ser completamente obtido a partir do outro, desde que se atue sobre o complemento da imagem utilizando o elemento estruturante refletido $\hat{B}$:

$$(A \ominus B)^c = A^c \oplus \hat{B} \quad \Longleftrightarrow \quad A \ominus B = (A^c \oplus \hat{B})^c$$

De forma análoga, a **dilatação também pode ser obtida a partir da erosão**:

$$(A \oplus B)^c = A^c \ominus \hat{B} \quad \Longleftrightarrow \quad A \oplus B = (A^c \ominus \hat{B})^c$$

Em termos práticos, a erosão de um objeto pode ser obtida pela dilatação de seu complemento, seguida da complementação do resultado (e vice-versa). Na implementação do pacote `morph.py`, as versões didáticas `mm::ero0` e `mm::dil0` tornam explícita essa estrutura por meio de laços (*loops*), enquanto `mm::ero` e `mm::dil` delegam as operações ao OpenCV visando maior eficiência computacional.

i Condições de contorno e imagens finitas

Na morfologia matemática clássica, definida sobre um domínio infinito (tipicamente $\mathbb{Z}^2$), essa dualidade é exata. Em imagens digitais, entretanto, trabalha-se com matrizes finitas, e o resultado passa a depender da forma como são tratados os pixels localizados fora da imagem.
Para que as identidades de dualidade permaneçam válidas, o complemento deve ser definido em relação ao mesmo universo e as condições de contorno adotadas para a erosão e para a dilatação devem ser complementares entre si. Por exemplo, se a erosão assume que os pixels externos pertencem ao objeto (255), então a dilatação aplicada ao complemento deve assumir que esses mesmos pixels externos pertencem ao fundo (0).
Quando diferentes estratégias de preenchimento são utilizadas (replicação, reflexão, valor constante etc.), a dualidade teórica pode deixar de ser satisfeita exatamente nas regiões próximas às bordas da imagem.

Para ilustrar numericamente os operadores morfológicos e a dualidade erosão–dilatação, a Figura 4.4 apresenta uma imagem binária 10×10 processada com um elemento estruturante em formato de "L". Na implementação de `morph.py`, a origem de B é fixada no centro geométrico da máscara — posição $(1,1)$ para um *kernel* 3×3 — e deve corresponder a um elemento ativo para que a erosão se comporte corretamente (conforme discutido anteriormente). O elemento estruturante B_L definido a seguir satisfaz essa condição. A Figura 4.5 complementa a análise com um simulador interativo da erosão, permitindo visualizar o deslocamento do

elemento estruturante sobre a imagem e identificar as posições em que ele permanece completamente contido no objeto.

```
%%writefile tmp/fig_04_ero_dil_didatico.cpp
#define MM_OUT "tmp/fig_04_ero_dil_didatico.png"
#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);
    // Preenche A com os valores do array numpy
    int valoresA[10][10] = {
        {0,0,0,0,0,0,0,0,0,0},
        {0,0,0,1,1,1,0,0,0,0},
        {0,0,1,1,1,1,1,0,0,0},
        {0,1,1,1,1,1,1,1,0,0},
        {0,1,1,1,1,1,1,1,0,0},
        {0,1,1,1,1,1,1,0,0,0},
        {0,0,1,1,1,1,1,1,0,0},
        {0,0,0,1,1,1,1,0,0,0},
        {0,0,0,0,1,0,0,0,0,0},
        {0,0,0,0,0,0,0,0,0,0}
    };
    for (int y = 0; y < 10; y++) {
        for (int x = 0; x < 10; x++) {
            A.at(y, x) = valoresA[y][x] * 255;
        }
    }

    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)

    // Converte SE para imagem para visualização
    mm::Image B_L_img(3, 3);
    for (int y = 0; y < 3; y++) {
        for (int x = 0; x < 3; x++) {
            B_L_img.at(y, x) = (B_L.at(y, x) >= 0) ? 255 : 0;
        }
    }
    mm::Image B_hat_img(3, 3);
    for (int y = 0; y < 3; y++) {
        for (int x = 0; x < 3; x++) {
            B_hat_img.at(y, x) = (B_hat.at(y, x) >= 0) ? 255 : 0;
        }
    }

    std::cout << "Elemento estruturante B_L:" << "\n"; std::cout << mm::drawImg(B_L_img) <<
"\n";
    std::cout << "Elemento estruturante refletido B:" << "\n"; std::cout <<
mm::drawImg(B_hat_img) << "\n";

    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);

    // Verifica igualdade pixel a pixel
    bool igual = true;
    for (int i = 0; i < A.h * A.w; i++) {
        if (ero_c.data[i] != dil_Ac.data[i]) {
            igual = false;
            break;
        }
    }
    std::cout << "Dualidade (A ⊖ B) == A ⊕ B : " << (igual ? "True" : "False") << "\n";

    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:
255 255 255
255 255 255
255 255 255

Elemento estruturante refletido B:
255 255 255
255 255 255
255 255 255

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)")
```

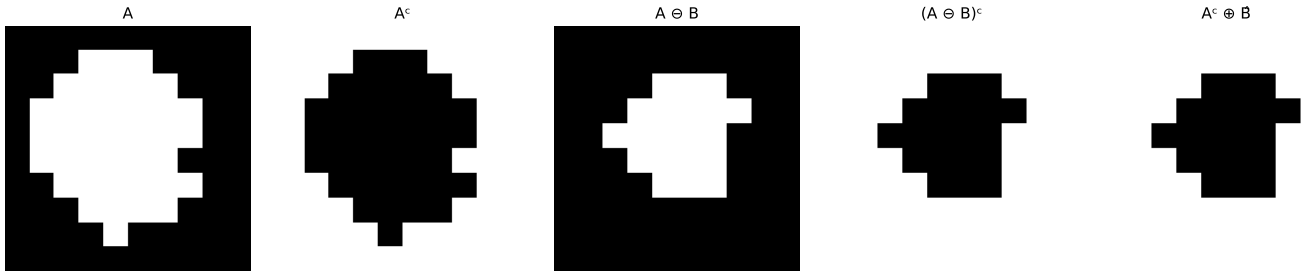


Figura 4.4: Erosão e dilatação em imagem binária 10×10 com elemento estruturante 'L' assimétrico 3×3. Validação da dualidade erosão–dilatação.

4.3.2 Abertura e Fechamento

Combinando erosão e dilatação obtêm-se dois operadores de grande utilidade prática: a **abertura** e o **fechamento**, definidos pelas Equações Equação 4.5 e Equação 4.6. Seus principais efeitos são resumidos na Tabela 4.1.

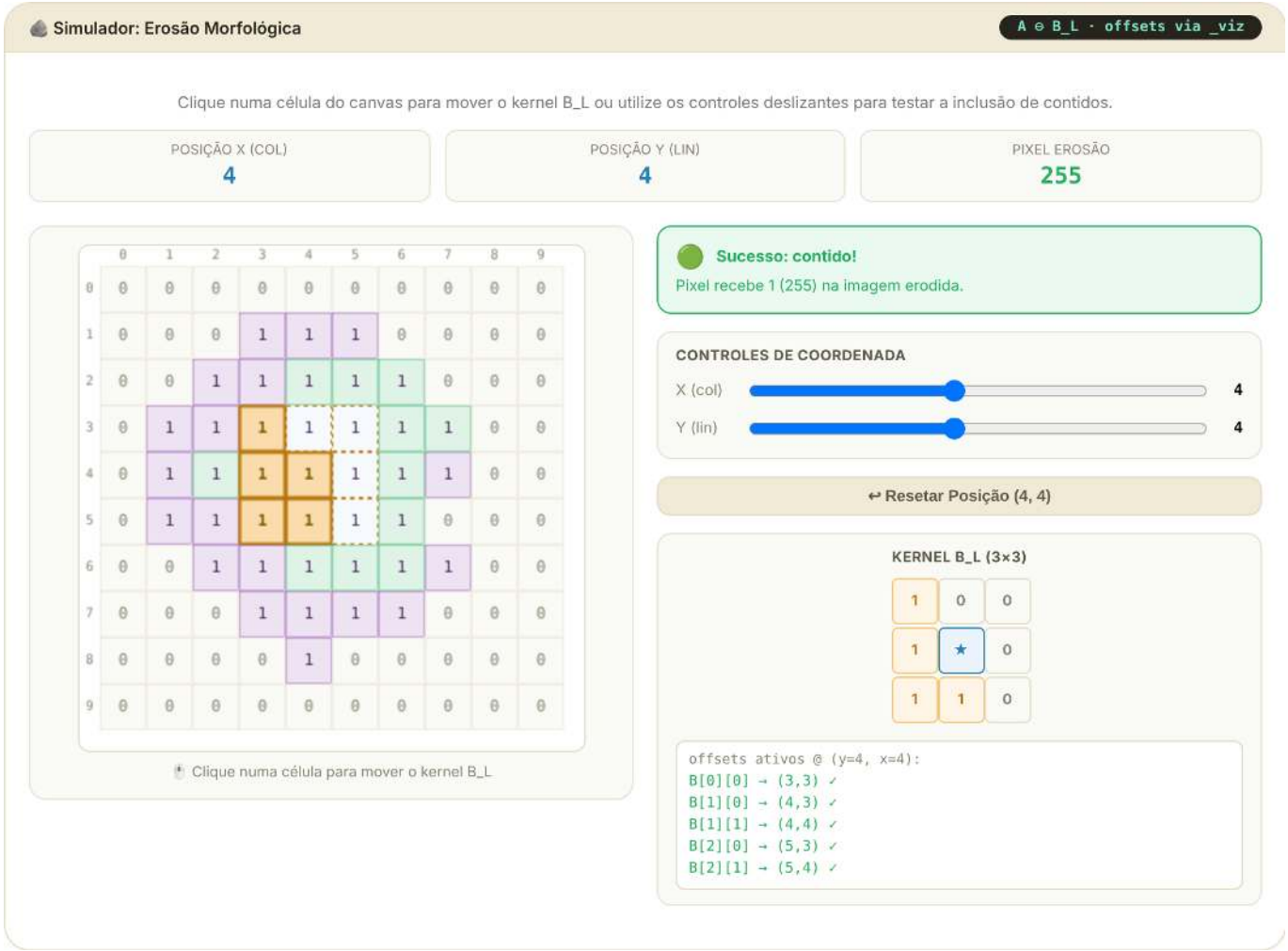
Abertura (*opening*) — erosão seguida de dilatação pelo mesmo B:

$$A \circ B = (A \ominus B) \oplus B \tag{4.5}$$

Fechamento (*closing*) — dilatação seguida de erosão pelo mesmo B:

$$A \bullet B = (A \oplus B) \ominus B \tag{4.6}$$

Na prática, `mm::open` e `mm::close` aplicam o mesmo elemento estruturante nas duas etapas. Para elementos estruturantes simétricos (os mais comuns), essa implementação coincide com a definição matemática apresentada acima.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-04-erosao.html

Figura 4.5: Simulador: Erosão Morfológica (A ⊖ B_L)

Tabela 4.1: Propriedades de abertura e fechamento.

Operador	Sequência	Efeito principal
Abertura $A \circ B$	erosão → dilatação	Remove estruturas incapazes de conter o elemento estruturante; suaviza contornos externos
Fechamento $A \bullet B$	dilatação → erosão	Preenche buracos menores que B; suaviza contornos internos

Propriedade importante: ambos são **idempotentes**. Por exemplo,

$$(A \circ B) \circ B = A \circ B,$$

ou seja, após a primeira aplicação, novas aplicações do mesmo operador não alteram mais o resultado.

4.3.2.1 Implementação da abertura e do fechamento

Diferentemente da erosão e da dilatação, abertura e fechamento não introduzem novos mecanismos computacionais. Ambos são obtidos pela composição sequencial dos operadores primitivos já apresentados:

```
def open0(f, B):
    return mm.dil0(mm.ero0(f, B), B)

def close0(f, B):
    return mm.ero0(mm.dil0(f, B), B)
```

A função `mm::open` delega a operação para `mm::open(f, B)`, enquanto `mm::close` utiliza `mm::close(f, B)`, produzindo o mesmo resultado de forma mais eficiente.

A abertura herda da erosão a capacidade de remover estruturas menores que o elemento estruturante e da dilatação a restauração parcial das regiões preservadas. O fechamento realiza o processo inverso: primeiro expande os objetos e depois restaura suas dimensões originais, preenchendo lacunas e buracos menores que o elemento estruturante.

4.3.2.2 Filtro Sequencial Alternado

Na prática, abertura e fechamento são frequentemente aplicados em sequência para remover simultaneamente ruído externo e preencher buracos internos. A função `mm::asf` (*Alternating Sequential Filter*) generaliza essa estratégia ao aplicar aberturas e fechamentos alternadamente com elementos estruturantes progressivamente maiores. As sequências disponíveis são apresentadas na Tabela 4.2.

Tabela 4.2: Sequências do filtro sequencial alternado mm.asf.

Sequência	Ordem	Uso típico
`'OC'`	abertura → fechamento	remove ruído externo antes de preencher pequenos buracos
`'CO'`	fechamento → abertura	preenche pequenos buracos antes de remover ruído externo
`'OCO'`	abertura → fechamento → abertura	enfatiza a remoção de ruído externo
`'COC'`	fechamento → abertura → fechamento	enfatiza o preenchimento de buracos e lacunas

O parâmetro `n` controla o número de escalas utilizadas pelo filtro. Em cada iteração i, o elemento estruturante é ampliado por soma de Minkowski (`mm::sesum(b, i)`), produzindo uma sequência de filtros morfológicos

cada vez mais abrangentes. Diferentemente de uma única abertura ou fechamento com um elemento estruturante grande, o ASF realiza uma suavização progressiva em múltiplas escalas, preservando melhor a geometria dos objetos relevantes enquanto elimina estruturas menores. A Figura 4.6 apresenta um exemplo de aplicação da abertura, do fechamento e do ASF na imagem das moedas.

```
%%writefile tmp/fig_04_open_close.cpp
#define MM_OUT "tmp/fig_04_open_close.png"
//| label: fig-04-open-close
//| fig-cap: "Abertura, fechamento, composição e filtro sequencial alternado aplicados à
binarização Otsu das moedas (pré-processadas por realce de contraste). Elemento estruturante:
disco 13×13."
//| 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_clahe = mm::_read_state("tmp/state/img_clahe_12.png");
// [pdi:state-io:end]

    // Variables img_clahe is provided automatically

    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);              // erosão → dilatação
    mm::Image img_close = mm::close(img_bin, B_disk);            // dilatação → erosão
    mm::Image img_oc = mm::close(img_open, B_disk);              // abertura seguida de
fechamento
    mm::Image img_asf = mm::asf(img_bin, "OC", mm::sedisk(3), 7);
    // ASF: disco base 3×3, cresce a cada iteração

    mm::show(
        std::vector<mm::Image>{img_bin, img_open, img_close, img_oc, img_asf},
        MM_OUT,
        std::vector<std::string>{"Binarização Otsu", "Abertura (A B)", "Fechamento (A B)",
                                 "Abertura→Fechamento", "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] Binarização Otsu
[2] Abertura (A B)
[3] Fechamento (A B)
[4] Abertura→Fechamento
[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)")
```

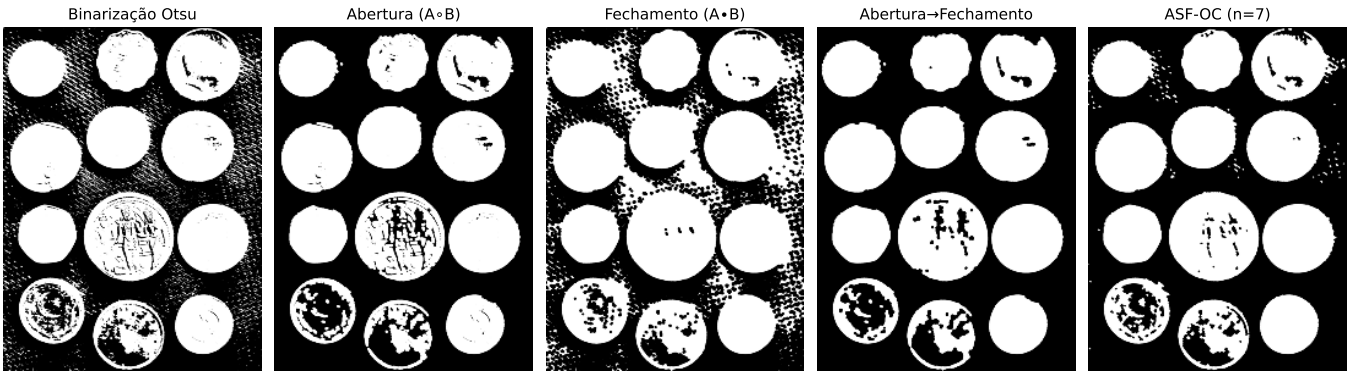


Figura 4.6: Abertura, fechamento, composição e filtro sequencial alternado aplicados à binarização Otsu das moedas (pré-processadas por realce de contraste). Elemento estruturante: disco 13×13.

4.3.3 Operadores Geodésicos

Os operadores geodésicos introduzem uma restrição adicional aos operadores morfológicos clássicos por meio de uma imagem de controle denominada **máscara** g. Em vez de permitir que a erosão ou a dilatação se propaguem livremente pela imagem, o resultado de cada iteração é limitado ponto a ponto pelos valores da máscara, restringindo a evolução da operação às regiões permitidas.

4.3.3.1 Dilatação Geodésica

A **dilatação geodésica** de uma imagem marcador f sob uma imagem máscara g, utilizando um elemento estruturante plano b, é definida por:

$$f \oplus_g b = (f \oplus b) \wedge g, \tag{4.7}$$

onde $\wedge$ representa o mínimo ponto a ponto.

Em outras palavras, realiza-se inicialmente uma dilatação convencional sobre o marcador e, em seguida, o resultado é restringido pela máscara g. Dessa forma, a propagação nunca pode ultrapassar as regiões permitidas pela máscara.

A formulação clássica da dilatação geodésica pressupõe que o marcador esteja contido na máscara, isto é, $f \leq g$, garantindo que a evolução da operação permaneça sempre limitada pela máscara.

4.3.3.2 Implementação da dilatação geodésica

A função `mm::cdil` implementa diretamente esse operador e permite executar múltiplas iterações consecutivas:

```
def cdil(f, g, b=np.zeros((3,3),dtype='uint8'), n=1):
    """Dilatação geodésica do marcador f sob a máscara g."""
    y = f.copy()
    for _ in range(n):
        y = np.minimum(mm.dil(y, b), g)
    return y
```

A instrução `np.minimum(mm::dil(y, b), g)` implementa exatamente a definição matemática da dilatação geodésica, isto é, $(y \oplus b) \wedge g$.

Quando $n = 1$, a função executa uma única dilatação geodésica. Para $n > 1$, o resultado de cada etapa torna-se o marcador da etapa seguinte, produzindo uma propagação progressiva controlada pela máscara.

O simulador interativo Figura 4.7 permite acompanhar, passo a passo, a propagação do marcador f ao longo dos corredores do labirinto. A cada iteração de `mm::cdil`, a frente de dilatação avança para as células vizinhas livres — aquelas em que $g = 1$ —, enquanto as paredes ($g = 0$) permanecem intransponíveis. O número de passos necessários para que o marcador alcance a saída corresponde exatamente ao comprimento geodésico do caminho mais curto dentro da máscara, evidenciando a conexão direta entre a dilatação geodésica iterada e a noção de distância em grafos.

4.3.3.3 Erosão Geodésica

De forma dual, a **erosão geodésica** de uma imagem marcador f sob uma imagem máscara g é definida por:

$$f \ominus_g b = (f \ominus b) \vee g, \tag{4.8}$$

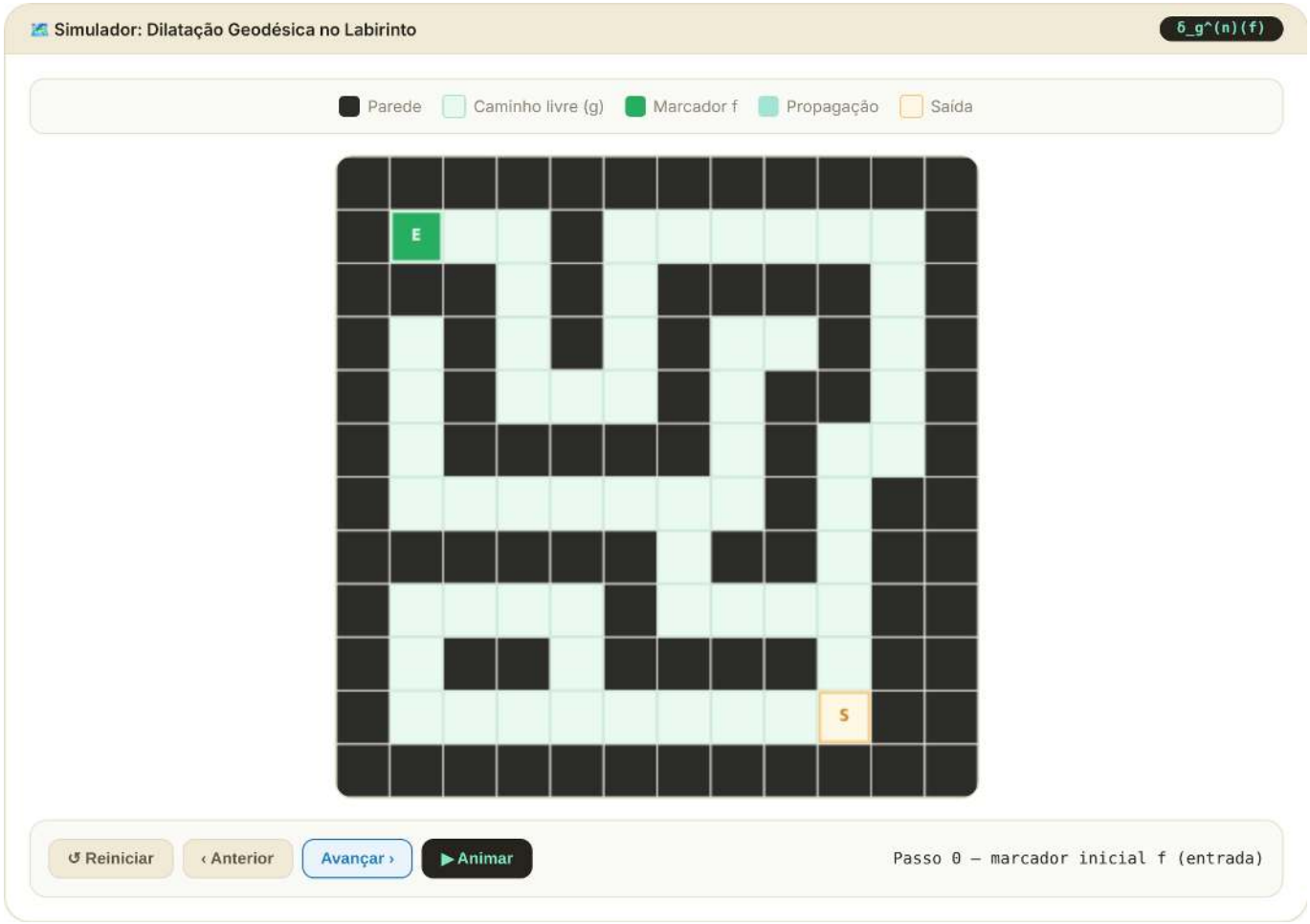
onde $\vee$ representa o operador de máximo ponto a ponto.

Nesse caso, a erosão convencional do marcador é seguida por uma restrição inferior imposta pela máscara. Assim, nenhum pixel do resultado pode assumir valor inferior ao correspondente pixel da máscara.

A formulação clássica da erosão geodésica pressupõe a condição dual

$$f \geq g,$$

de modo que a máscara atue como limite inferior durante todo o processo.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-04-cdil.html

Figura 4.7: Simulador interativo de dilatação geodésica: o marcador f (verde) propaga-se passo a passo pelos caminhos livres da máscara g, sem atravessar paredes.

4.3.3.4 Implementação da erosão geodésica

A função estática `mm::cero` materializa esse operador:

```
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
```

A instrução `np.maximum(mm::ero(y, b), g)` implementa diretamente a expressão $(y \ominus b) \vee g$.

Assim como na dilatação geodésica, o parâmetro n define quantas erosões geodésicas sucessivas serão computadas antes de retornar a imagem final.

> **i** Relação com a reconstrução morfológica
>
> A reconstrução morfológica apresentada na próxima seção é obtida pela aplicação iterativa da dilatação geodésica (`mm::cdil`) até que um ponto fixo seja alcançado, isto é, até que nenhuma célula mude de valor entre duas iterações consecutivas. Em outras palavras, a reconstrução consiste em uma sequência de dilatações geodésicas sucessivas que se propagam dentro da máscara até não haver mais alterações. De forma dual, também é possível definir reconstruções baseadas em erosão geodésica por meio de aplicações sucessivas de `mm::cero`.

4.3.3.5 Exemplo: propagação em um labirinto via dualidade

A Figura 4.8 ilustra a resolução do problema de conectividade de um labirinto utilizando a dualidade morfológica por meio das funções `mm::cero` e `mm::suprec`.

Em vez de propagar um marcador pelos corredores livres através de dilatações geodésicas, o problema é formulado no domínio complementar. Inicialmente, a máscara original é invertida,

$$g = 1 - g_{orig},$$

fazendo com que as paredes passem a assumir valor 1 e os corredores valor 0. De forma análoga, o marcador é construído nesse mesmo domínio complementar, contendo um único valor 0 na posição de entrada do labirinto e valor 1 nos demais pixels.

Utilizando o elemento estruturante em cruz (`mm::secross()`), a erosão geodésica atua sobre o marcador complementado. A cada iteração de `mm::cero`, a região conectada ao marcador inicial sofre erosões sucessivas, enquanto a máscara impõe um limite inferior que impede a propagação através das paredes do labirinto.

As imagens intermediárias mostram estados da evolução após diferentes números de iterações (`n=5`, `n=12` e `n=22`). O resultado final é obtido pela reconstrução geodésica por erosão (`mm::suprec`), que aplica erosões geodésicas sucessivas até atingir um ponto fixo, isto é, uma situação em que nenhuma alteração adicional ocorre entre duas iterações consecutivas.

No domínio complementar, a região reconstruída corresponde exatamente ao conjunto de corredores conectados à entrada do labirinto. Assim, a conectividade entre entrada e saída pode ser determinada diretamente a partir da imagem reconstruída.

```
%%writefile tmp/fig_04_cero_labirinto.cpp
#define MM_OUT "tmp/fig_04_cero_labirinto.png"
//| 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

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

int main() {

    // Máscara já invertida (255 = parede, 0 = corredor)
    mm::Image g(10, 10);
    {
        int vals[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) = (unsigned char)vals[y][x];
    }

    mm::Image f(10, 10);
    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)")
```

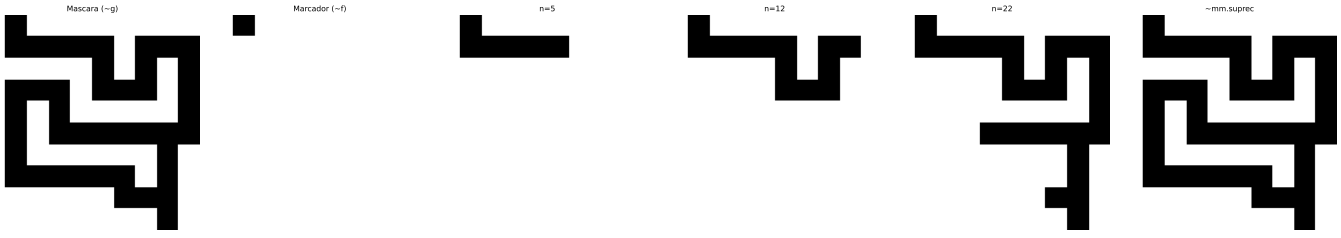


Figura 4.8: Resolução de labirinto no domínio complementar: com máscara e marcador invertidos, *mm::cero* e *mm::suprec* propagam a onda geodésica pelos corredores.

4.3.4 Reconstrução Morfológica

A **reconstrução morfológica** propaga uma imagem **marcador** f dentro de uma imagem **máscara** g, garantindo que o resultado nunca ultrapasse os valores de intensidade impostos pela máscara. O operador fundamental que viabiliza essa propagação contida é a **dilatação geodésica**, definida pela Equação 4.7.

A reconstrução é obtida pela aplicação iterativa dessa dilatação condicionada. Inicialmente, o marcador é limitado pela máscara para estabelecer o estado inicial:

$$X^{(0)} = f \wedge g,$$

e as iterações subsequentes são definidas de forma recursiva por:

$$X^{(k)} = (X^{(k-1)} \oplus b) \wedge g.$$

A sequência cresce monotonicamente até atingir um ponto fixo, produzindo a **reconstrução morfológica por dilatação** (também conhecida na literatura como *inf-reconstrução*):

$$R_g^{\delta}(f) = \lim_{k \to \infty} X^{(k)} = X^{(k)} \quad \text{quando} \quad X^{(k)} = X^{(k-1)}. \tag{4.9}$$

A subida iterativa é interrompida assim que a estabilidade é alcançada, isto é, quando duas iterações consecutivas produzem matrizes com valores absolutamente idênticos.

4.3.4.1 Implementação da reconstrução morfológica

A rotina didática `mm::infrec` implementa diretamente o algoritmo iterativo de ponto fixo. Inicialmente, o marcador efetivo inicial $X^{(0)}$ é determinado pela operação `np.minimum(f, g)`. Para garantir que o laço de verificação execute a primeira passada sem disparar falsas convergências prematuras, a variável de controle da iteração anterior (`y1`) é inicializada preenchida com um valor sentinela fora do domínio de dados (ou simplesmente com uma matriz que force a primeira execução).

```
def infrec(f, g, b=np.zeros((3,3), dtype='uint8')):
    """Inf-reconstrução: dilata o marcador (f   g) até convergir sob a máscara g."""
    y = np.minimum(f, g)
    # Inicializa y1 com valores impossíveis para forçar a entrada no laço
    y1 = np.full_like(f, 256, dtype=np.int16)
    while not np.array_equal(y, y1):
        y1 = y.copy()
        # Aplica a dilatação geodésica: (y   b)   g
        y = np.minimum(mm.dil(y, b), g)
    return y.astype('uint8')
```

No interior do laço `while`, a variável `y` armazena a estimativa corrente da reconstrução $X^{(k)}$, enquanto `y1` preserva a imagem do estágio imediatamente anterior $X^{(k-1)}$. A instrução de controle condicional `np.minimum(mm::dil(y, b), g)` traduz fielmente a dilatação geodésica teórica, onde a expansão morfológica convencional comandada pelo OpenCV é imediatamente "podada" e limitada pelas barreiras de intensidade da máscara g. O laço cessa quando nenhuma modificação de pixel é registrada entre os passos.

4.3.4.2 Vantagens da reconstrução morfológica

A reconstrução morfológica é significativamente mais robusta que a abertura convencional porque é capaz de remover estruturas indesejadas sem distorcer ou alterar a morfologia dos objetos que devem ser preservados.

Enquanto a abertura clássica suaviza cantos, elimina pontas e deforma contornos devido à imposição geométrica rígida do elemento estruturante, a reconstrução geodésica utiliza a máscara para recuperar com exatidão os limites e formatos originais dos objetos que possuem conectividade com o marcador original.

Em termos intuitivos, o marcador atua como uma semente de contágio que se expande progressivamente, mas apenas trafegando pelas regiões permitidas pela máscara. Componentes que não possuem nenhuma intersecção com o marcador jamais serão reconstruídas (sendo eliminadas), enquanto as componentes tocadas pela semente expandem-se até restaurar integralmente a sua geometria original.

Esse comportamento discriminatório e conservativo é ilustrado na Figura 4.9, utilizando o elemento estruturante em cruz apresentado na Figura 4.3.

```
%%writefile tmp/fig_04_reconstrucao_didatica.cpp
#define MM_OUT "tmp/fig_04_reconstrucao_didatica.png"
//| 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
#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>

int main() {
    mm::Image g(10, 10);
    int vals[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) = vals[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 < 6; 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);
    bool estavel = true;
    for (size_t i = 0; i < img_reconstruida.data.size(); i++) {
        if (img_reconstruida.data[i] != iteracoes.back().data[i]) {
            estavel = false;
            break;
        }
    }
    std::cout << "  Estabilidade na iteração 5: " << (estavel ? "True" : "False") << "\n";

    std::vector<mm::Image> todas = {g, f};
    todas.insert(todas.end(), iteracoes.begin(), iteracoes.end());
    todas.push_back(img_reconstruida);

    std::vector<std::string> todos_titulos = {"Máscara (g)", "Marcador (f)"};
    todos_titulos.insert(todos_titulos.end(), titulos.begin(), titulos.end());
    todos_titulos.push_back("Reconstrução R_g(f)");

    mm::show(todas, MM_OUT, todos_titulos, 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)")
```

Figura 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 Preenchimento de Buracos e Remoção de Bordas

Dois operadores baseados em reconstrução morfológica completam o *pipeline* de limpeza binária. Suas características principais são resumidas na Tabela 4.3.

Preenchimento de buracos (`mm::clohole`) remove cavidades completamente cercadas pelo objeto, independentemente do tamanho, sem alterar os contornos externos. O procedimento atua no complemento da imagem, utilizando como marcador uma restrição da moldura (*frame*) ao fundo:

$$\mathrm{clohole}(f) = \left(R^{\delta}_{f^c}(\mathrm{frame}(f) \wedge f^c)\right)^c \tag{4.10}$$

Em termos operacionais, primeiro reconstrói-se o fundo externo e, em seguida, aplica-se a complementação para recuperar os objetos com buracos preenchidos.

Remoção de objetos de borda (`mm::edgeoff`) elimina todos os objetos que tocam a borda da imagem, preservando apenas os componentes totalmente internos. O marcador é obtido pela interseção entre a moldura (*frame*) e os objetos da imagem:

$$\text{edgeoff}(f) = f \setminus R_f^{\delta}(\text{frame}(f) \wedge f) \tag{4.11}$$

Tabela 4.3: Comparação entre os operadores *clohole* e *edgeoff*.

Operador	Marcador	Máscara	Efeito
`mm::clohole`	*frame* restrito ao fundo (f^c)	f^c	Preenche buracos internos
`mm::edgeoff`	*frame* restrito ao objeto (f)	f	Remove objetos conectados à borda

A evolução passo a passo dessas transformações geodésicas pode ser acompanhada nas figuras a seguir. A Figura 4.10 ilustra o mecanismo de inundação controlada do operador `mm::clohole`, no qual a reconstrução ocorre a partir do fundo externo e impede a propagação para regiões internas não conectadas ao exterior, resultando no preenchimento consistente das cavidades internas. Em contrapartida, a Figura 4.11 detalha a dinâmica do operador `mm::edgeoff`, em que apenas os componentes conectados à borda são reconstruídos e posteriormente removidos, preservando exclusivamente os objetos totalmente contidos no interior da imagem.

A conectividade da propagação geodésica é controlada pelo elemento estruturante: `mm::sebox()` (vizinhança de 8) inclui conexões diagonais, enquanto `mm::secross()` (vizinhança de 4) as exclui. Consequentemente, a escolha do elemento estruturante afeta quais componentes são alcançados pela reconstrução e, portanto, quais serão preservados ou removidos.

4.3.5.1 Conformidade com a implementação

As definições acima estão diretamente alinhadas com a implementação em `morph.py`, reproduzida a seguir:

```
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))
```

Essas implementações deixam explícito que ambos os operadores são instâncias diretas de reconstrução morfológica por dilatação geodésica com `mm::infrec`, diferindo apenas na escolha do marcador e da máscara: `clohole` atua no complemento da imagem, enquanto `edgeoff` atua diretamente no domínio dos objetos.

```
%%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>

int main() {
    // Criar imagem f a partir do array numpy
```

```
    mm::Image f(10, 10);
    // Matriz de entrada (0 e 1, depois multiplicada por 255)
    int dados[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) = dados[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));

    // Verificação: comparar com mm::clohole(f)
    mm::Image clohole_ref = mm::clohole(f);
    bool igual = true;
    for (int i = 0; i < (int)img_clohole.data.size(); i++) {
        if (img_clohole.data[i] != clohole_ref.data[i]) {
            igual = false;
            break;
        }
    }
    std::cout << "  Validação clohole: " << (igual ? "True" : "False") << "\n";

    std::vector<mm::Image> todas;
    todas.push_back(f);
    todas.push_back(marcador_ch);
    todas.insert(todas.end(), iteracoes.begin(), iteracoes.end());
    todas.push_back(img_clohole);

    std::vector<std::string> todos_titulos;
    todos_titulos.push_back("f original");
    todos_titulos.push_back("Marcador (borda)");
    todos_titulos.insert(todos_titulos.end(), titulos.begin(), titulos.end());
    todos_titulos.push_back("clohole(f)");

    mm::show(todas, MM_OUT, todos_titulos, 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)")
```

Figura 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"
//| label: fig-04-edgeoff-didatico
//| fig-cap: "*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."
//| echo: true
//| output: true

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

int main() {
    mm::Image f(10, 10, 1);
    int data[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) = (unsigned char)(data[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 <= 5; 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> images;
    images.push_back(f);
    images.push_back(marcador_eo);
    for (auto& img : iteracoes) images.push_back(img);
    images.push_back(img_edgeoff);

    std::vector<std::string> titles;
    titles.push_back("f original");
    titles.push_back("Marcador (borda)");
    for (auto& t : titulos) titles.push_back(t);
    titles.push_back("edgeoff(f)");

    mm::show(images, MM_OUT, 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)")
```

Figura 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 *Pipeline* de Limpeza Binária com CLAHE

Com base na análise anterior — na qual o CLAHE produziu o maior valor da variância interclasses ($\sigma_B^2 \approx 2{,}47 \times 10^3$), ver Figura 4.2, e o operador `mm::clohole` mostrou-se eficaz no preenchimento das cavidades internas —, o *pipeline* final de segmentação, ilustrado na Figura 4.12, é estruturado pelo seguinte fluxo computacional:

$$\text{gray} \xrightarrow{\text{CLAHE}} \xrightarrow{\text{Otsu}} \xrightarrow{\text{open}} \xrightarrow{\text{clohole}} \xrightarrow{\text{open}} \xrightarrow{\text{edgeoff}} \text{segmentação}$$

Após a etapa de `mm::clohole`, aplica-se uma segunda abertura morfológica com um elemento estruturante maior (`mm::sedisk(33)`, disco de diâmetro 33). Essa operação remove pequenas regiões residuais e artefatos que possam ter permanecido após a segmentação. Em particular, o preenchimento geodésico pode transformar pequenas cavidades isoladas em componentes conectados ao objeto, tornando conveniente uma etapa adicional de filtragem baseada em tamanho. O diâmetro foi escolhido de modo que as moedas permaneçam capazes de conter o elemento estruturante, enquanto componentes significativamente menores sejam eliminados.

Nesta imagem, nenhuma moeda está conectada à borda da matriz. Consequentemente, a aplicação de `mm::edgeoff` não altera o resultado obtido após a segunda abertura. Ainda assim, essa etapa é mantida no *pipeline* por robustez, pois em outras imagens podem existir objetos parcialmente visíveis ou conectados às bordas, que devem ser removidos antes da etapa de análise.

Por que a abertura após o clohole?

O operador `mm::clohole` preenche todas as cavidades fechadas presentes nos objetos segmentados. Em algumas situações, pequenas regiões indesejadas podem permanecer após essa etapa ou tornar-se conectadas aos objetos principais. A abertura morfológica subsequente remove componentes menores que o elemento estruturante, preservando as moedas devido ao seu tamanho significativamente maior.

```
%%writefile tmp/fig_04_pipeline_clahe.cpp
#define MM_OUT "tmp/fig_04_pipeline_clahe.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_coins_gray = mm::_read_state("tmp/state/img_coins_gray_8.png");
// [pdi:state-io:end]

    //   Pré-processamento: apenas CLAHE
```

```
    mm::Image img_clahe0 = mm::clahe(img_coins_gray, 2.0, 8);

    //   Etapa 1: Binarização Otsu
    mm::Image img_bin = mm::threshold(img_clahe0);

    //   Etapa 2: Abertura - remove ruídos brancos de fundo
    mm::Image img_open = mm::open(img_bin, mm::sedisk(9));

    //   Etapa 3: clohole - fecha todos os buracos internos
    mm::Image img_hole = mm::clohole(img_open);

    //   Etapa 4: Abertura (kernel grande) - remove artefatos do clohole
    mm::Image img_limpo = mm::open(img_hole, mm::sedisk(33));

    //   Etapa 5: edgeoff - remove objetos que tocam a borda
    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
    );

    // img_final is guaranteed to hold the final computed value here

// [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)")
```

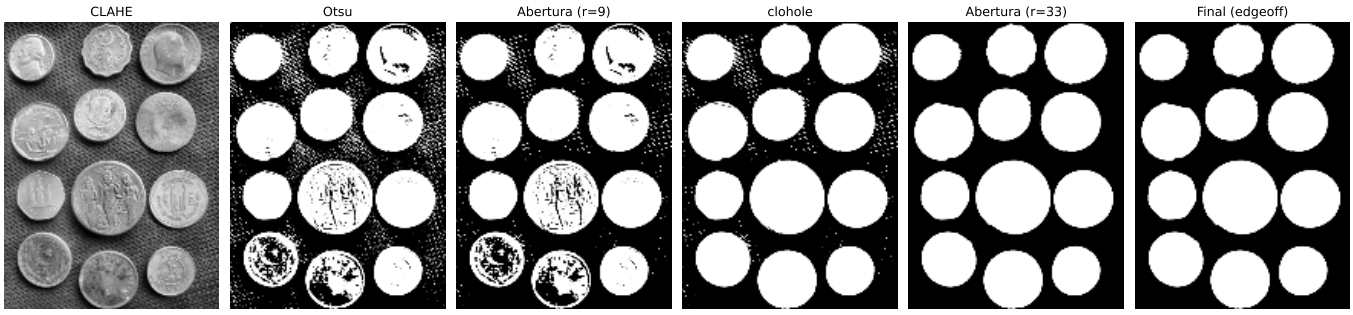


Figura 4.12: *Pipeline* completo de segmentação com CLAHE: Otsu → abertura (r=9) → clohole → abertura (r=33) → edgeoff.

4.3.7 Morfologia em Tons de Cinza

Os operadores morfológicos estendem-se naturalmente para imagens em tons de cinza. Nessa formulação, a erosão e a dilatação passam a atuar diretamente sobre os níveis de intensidade da imagem. Para elementos estruturantes planos ($b \equiv 0$), a erosão corresponde ao **mínimo local** e a dilatação ao **máximo local** dentro da vizinhança definida pelo elemento estruturante.

A interpretação intuitiva é simples: a erosão **escurece** regiões ao substituir cada pixel pelo menor valor presente em sua vizinhança, enquanto a dilatação **clareia** regiões ao utilizar o maior valor disponível. A combinação desses operadores permite construir transformações capazes de realçar bordas, remover tendências de iluminação e destacar estruturas locais.

Três operadores derivados são especialmente úteis:

Gradiente morfológico — destaca bordas como a diferença entre dilatação e erosão:

$$\mathrm{grad}_B(f) = (f \oplus B) - (f \ominus B) \tag{4.12}$$

Top-hat — destaca estruturas brilhantes menores que o elemento estruturante (diferença entre a imagem original e sua abertura):

$$\text{top-hat}_B(f) = f - (f \circ B) \tag{4.13}$$

Black-hat — destaca estruturas escuras menores que o elemento estruturante (diferença entre o fechamento e a imagem original):

$$\text{black-hat}_B(f) = (f \bullet B) - f \tag{4.14}$$

O *Top-hat* extrai detalhes brilhantes que não sobrevivem à abertura, enquanto o *Black-hat* evidencia detalhes escuros removidos pelo fechamento. Já o gradiente morfológico realça transições abruptas de intensidade, produzindo uma representação semelhante à de um detector de bordas.

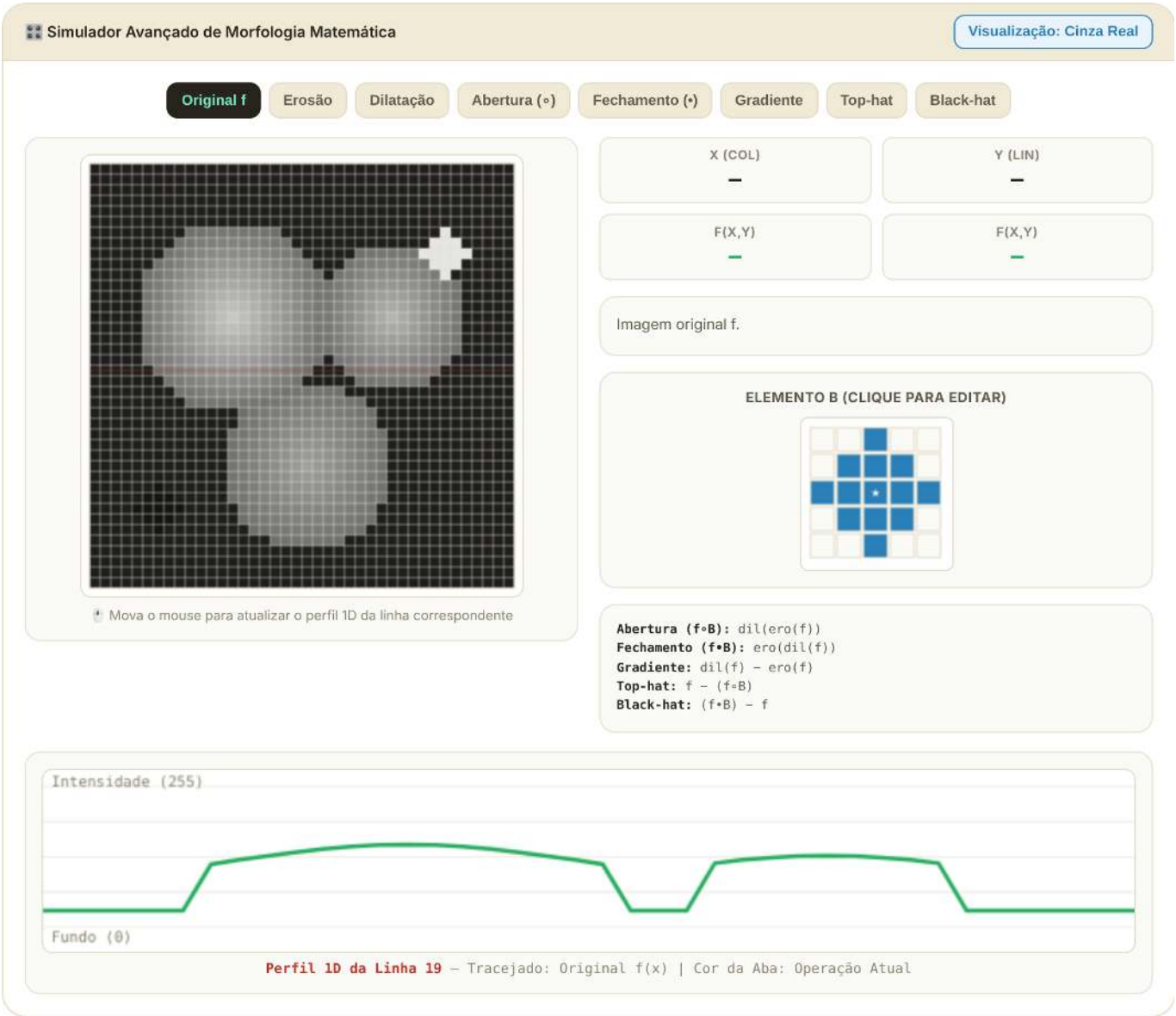
Para compreender o mecanismo desses operadores em nível local, a Figura 4.13 apresenta um simulador interativo de morfologia em tons de cinza. O simulador permite editar livremente o elemento estruturante, visualizar seu deslocamento sobre a imagem e acompanhar simultaneamente o perfil unidimensional das intensidades. Dessa forma, torna-se possível observar diretamente como a erosão seleciona mínimos locais, como a dilatação seleciona máximos locais e como o gradiente morfológico emerge da diferença entre esses dois operadores.

O botão localizado no canto superior direito permite alternar entre a visualização original em tons de cinza e uma representação pseudocolorida (*colormap*) apenas nos três tipos gradientes. A versão colorida facilita a percepção visual das variações de intensidade, tornando mais evidente a ação dos operadores morfológicos sobre máximos, mínimos e transições locais da imagem.

Os operadores apresentados estão disponíveis em `morph.py` por meio das funções `mm::gradm`, `mm::tophat` e `mm::blackhat`.

A Figura 4.14 ilustra os efeitos desses operadores sobre a imagem das moedas e seus respectivos histogramas. Observe que a erosão desloca a distribuição para intensidades mais baixas, enquanto a dilatação a desloca para intensidades mais altas. O gradiente concentra valores nas regiões de contorno, e os operadores *Top-hat* e *Black-hat* produzem histogramas fortemente concentrados em baixos níveis de intensidade, pois apenas pequenas estruturas locais são realçadas.

```
%%writefile tmp/fig_04_morf_gc_histogramas.cpp
#define MM_OUT "tmp/fig_04_morf_gc_histogramas.png"
//| 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

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

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]

    mm::SE B = mm::sedisk(19);  // disco de diâmetro 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);

    mm::show(
        {img_coins_gray, mm::histImg(img_coins_gray), img_ero, mm::histImg(img_ero),
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-04-operadores-av.html

Figura 4.13: Simulador interativo avançado de morfologia com elemento estruturante editável e perfil 1D.

```
         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,
        {"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)")
```

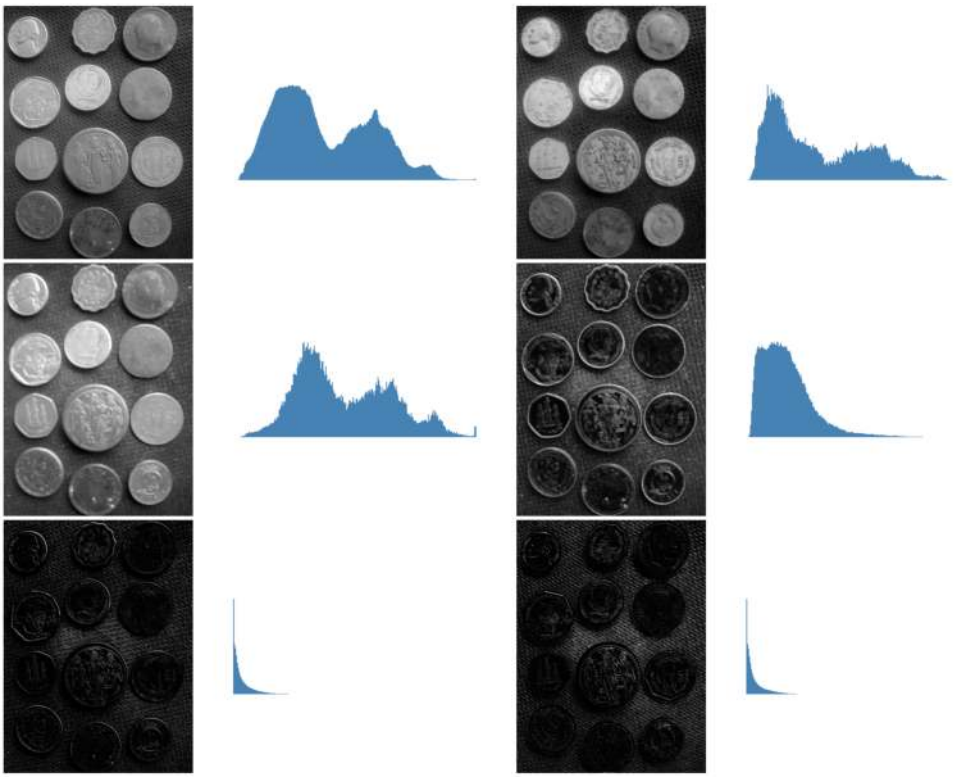


Figura 4.14: Morfologia em tons de cinza e seus histogramas: erosão, dilatação, gradiente, top-hat e black-hat. Elemento estruturante: disco de diâmetro 19.

Os operadores morfológicos apresentados anteriormente serão agora utilizados como ferramentas de refinamento e geração de marcadores para métodos de segmentação mais avançados, apresentados a seguir.

4.4 Segmentação de Imagens: Fundamentação e Taxonomia

A **segmentação de imagens** consiste em dividir a imagem em regiões associadas a objetos ou estruturas de interesse. Em PDI, ela representa a transição entre o processamento de baixo nível — como filtragem e realce — e etapas de análise mais avançadas, como extração de características, reconhecimento e interpretação da cena.

Formalmente, o objetivo da segmentação consiste em decompor o domínio espacial completo de uma imagem, denotado por Ω, em uma partição de subconjuntos $\{R_1, R_2, \dots, R_n\}$ que satisfaça simultaneamente os critérios de **completeza** e **disjunção**:

$$\bigcup_{i=1}^{n} R_i = \Omega, \qquad R_i \cap R_j = \emptyset \quad \forall\, i \neq j \tag{4.15}$$

Além das propriedades de completeza e disjunção expressas na Equação 4.15, cada sub-região R_i deve constituir um domínio **homogêneo** segundo um predicado de similaridade definido sobre propriedades locais — intensidade, cor ou textura — e, simultaneamente, ser **distinta** das regiões adjacentes.

As técnicas de segmentação podem ser organizadas em diferentes famílias. Neste capítulo serão enfatizadas as abordagens resumidas na Tabela 4.4, fundamentadas principalmente em critérios de intensidade, conectividade e proximidade espacial.

Tabela 4.4: Taxonomia simplificada das principais abordagens de segmentação e refinamento estudadas neste capítulo.

Abordagem	Critério de Segmentação	Operadores de Referência
Limiarização	Particionamento do espaço de intensidades	Critério de Otsu, limiarização global e local
Morfologia Matemática	Relações espaciais definidas por elementos/funções estruturantes	Erosão, dilatação, abertura, fechamento e reconstrução
Baseada em Regiões	Homogeneidade local e conectividade espacial	Rotulação de componentes conexos, Transformada de Distância e *Watershed*

Até este ponto, o desenvolvimento prático concentrou-se na **limiarização**, por meio da combinação entre equalização adaptativa CLAHE e o método global de Otsu. Essa etapa foi complementada por operadores de **reconstrução morfológica** baseados em dilatações geodésicas, implementados pelas funções `mm::infrec`, `mm::clohole` e `mm::edgeoff`, produzindo uma máscara binária limpa e adequada para análise.

Contudo, em cenários onde objetos distintos aparecem conectados na máscara binária — seja por contato físico, sobreposição parcial ou por pontes estreitas de pixels produzidas pela segmentação —, a limiarização deixa de ser suficiente para individualizar cada objeto. Nesses casos, múltiplos objetos passam a compor um único componente conexo, dificultando etapas posteriores de medição e interpretação.

Para superar essa limitação, as próximas seções introduzem três ferramentas complementares: a **Rotulação de Componentes Conexos**, a **Transformada de Distância** e o algoritmo de segmentação por ***Watershed*** baseado em marcadores. Em conjunto, essas técnicas permitem separar objetos adjacentes, identificar regiões individualmente e extrair descritores geométricos consistentes para análise quantitativa.

4.4.1 Rotulação

A **rotulação de componentes conexas** (*connected component labeling*) é o operador que atribui um identificador inteiro único a cada conjunto de pixels pertencentes à mesma componente conexa em uma imagem binária.

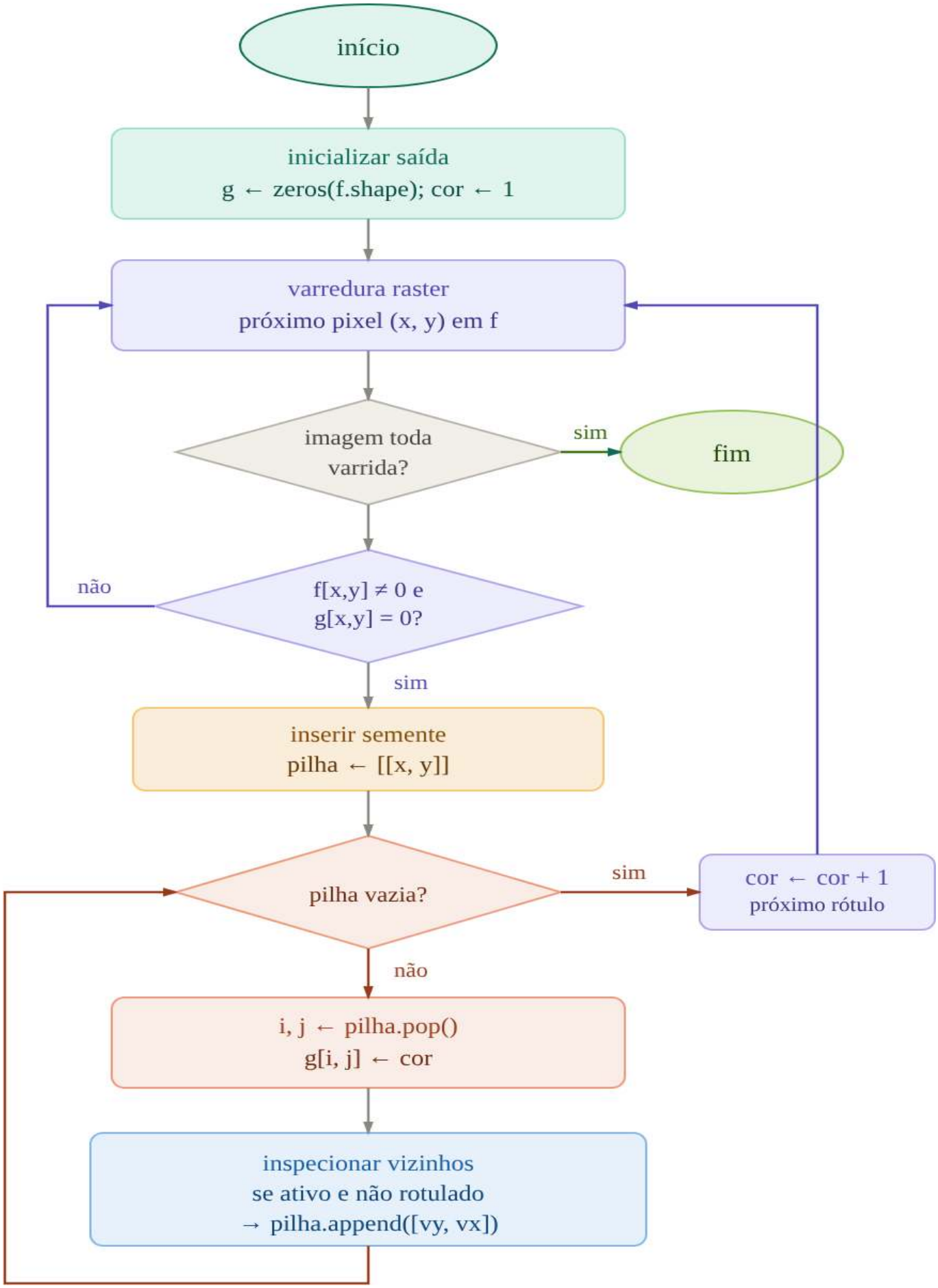
> **i** Definição formal
>
> Dada uma imagem binária f e uma relação de conectividade definida por um elemento estruturante B (tipicamente conectividade-4 ou conectividade-8), o algoritmo de rotulação da Figura 4.15 produz uma imagem g na qual todos os pixels pertencentes à mesma componente conexa recebem o mesmo rótulo inteiro positivo, enquanto pixels pertencentes a componentes distintas recebem rótulos diferentes.

A conectividade define quais pixels são considerados vizinhos diretos de um pixel (x, y). As definições mais utilizadas são:

- **Conectividade-4:** considera apenas os quatro vizinhos ortogonais (norte, sul, leste e oeste).
- **Conectividade-8:** considera os quatro vizinhos ortogonais e os quatro diagonais, totalizando oito vizinhos.

A escolha da conectividade influencia diretamente a formação das componentes conexas e, consequentemente, o resultado da rotulação, como ilustrado na Figura 4.17. Um exemplo adicional pode ser explorado interativamente no simulador apresentado na Figura 4.16.

A implementação em `morph.py` disponibiliza duas versões desse operador. A função `mm::label0` reproduz explicitamente o algoritmo de *flood-fill* utilizando uma pilha e permite controlar a conectividade por meio do elemento estruturante adotado. Já `mm::label` delega a operação à implementação otimizada do OpenCV (`mm::label0`). Em ambos os casos, o resultado é uma imagem rotulada na qual cada componente conexa recebe um identificador inteiro distinto.

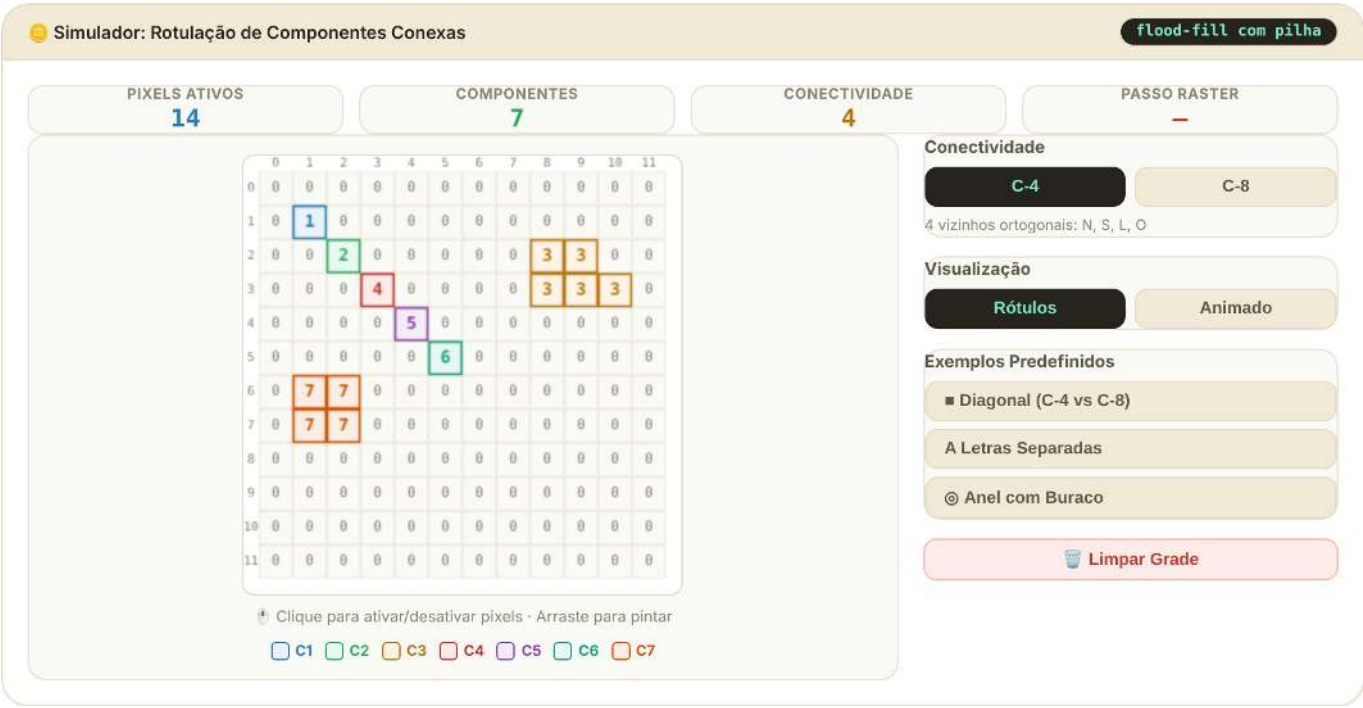


Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-alg-rotulagem2.html

Figura 4.15: Algoritmo de rotulação por *flood-fill* com pilha.

O auxiliar `_viz` itera sobre a janela estruturante `b` centrada em (i, j), gerando somente os vizinhos válidos dentro dos limites da imagem — a conectividade desejada é inteiramente determinada pela forma de `b` passada ao algoritmo.

Exemplo didático — efeito da conectividade:



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-04-rotulacao.html

Figura 4.16: Simulador interativo de rotulação de componentes conexas (*connected component labeling*): visualização da expansão flood-fill, conectividade-4 e conectividade-8.

```
%%writefile tmp/fig_04_rotulacao_didatico.cpp
#define MM_OUT "tmp/fig_04_rotulacao_didatico.png"
#include "morph.hpp"
#include <iostream>
#include <algorithm>

//| 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

int main() {

    mm::Image f(10, 10);
    // Preenchendo a matriz manualmente
    int vals[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) = vals[y][x] * 255;

    mm::Image B4 = mm::secross();   // conectividade-4
    mm::Image B8 = mm::sebox();     // conectividade-8

    mm::Image lbl4 = mm::label0(f, B4);
    mm::Image lbl8 = mm::label0(f, B8);

    // Encontrando os máximos (equivalente a lbl.max())
    int n4 = 0, n8 = 0;
    for (int i = 0; i < (int)lbl4.data.size(); i++)
        n4 = std::max(n4, (int)lbl4.data[i]);
    for (int i = 0; i < (int)lbl8.data.size(); i++)
        n8 = std::max(n8, (int)lbl8.data[i]);

    std::cout << "Componentes C4: " << n4 << "  |  C8: " << n8 << "\n";

    // Função norm_label
    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;
    };

    int mx4 = std::max(n4, 1);
    int mx8 = std::max(n8, 1);

    mm::Image nl4 = norm_label(lbl4, mx4);
    mm::Image nl8 = norm_label(lbl8, mx8);

    std::vector<std::string> titles = {
        "f original",
        "C4 (" + std::to_string(n4) + " comp.)",
        "C8 (" + std::to_string(n8) + " comp.)"
    };

    mm::show(std::vector<mm::Image>{f, nl4, nl8}, MM_OUT, titles, 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)")
```

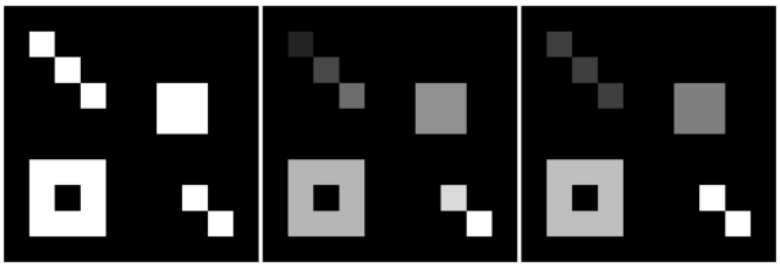


Figura 4.17: Efeito da conectividade na rotulação (*mm::label0*): pixels diagonalmente adjacentes formam componentes distintas em conectividade-4 e se fundem em conectividade-8.

4.4.2 Transformada de Distância

A **Transformada de Distância** (TD) é um operador que, aplicado a uma imagem binária f, produz uma imagem em níveis de cinza D na qual cada pixel pertencente ao objeto ($f(x, y) \neq 0$) recebe como valor a distância geométrica até o pixel de fundo ($f(x', y') = 0$) mais próximo:

$$D(x, y) = \min_{(x', y') \,:\, f(x', y') = 0} d\big((x, y),\, (x', y')\big)$$

onde $d(\cdot, \cdot)$ é uma métrica de distância — tipicamente a distância Euclidiana (L_2). O resultado é uma representação topográfica dos objetos: pixels localizados no interior assumem valores elevados, enquanto pixels próximos às bordas apresentam baixos valores de distância. Os **máximos locais** de D correspondem aos pontos mais afastados da borda do objeto, frequentemente próximos aos seus centros geométricos ou centros de máxima inscrição — propriedade particularmente útil para a geração automática de marcadores no algoritmo *watershed*.

> **i** Definição formal usando erosões
>
> A TD admite ainda uma interpretação morfológica iterativa, conforme algoritmo da Figura 4.18. Considere uma função estruturante b cujo valor central é nulo e cujos vizinhos possuem custos negativos associados ao deslocamento. Ao aplicar erosões sucessivas com essa função estruturante particular, os valores dos pixels dos objetos (que devem assumir a distância máxima possível da imagem) são progressivamente reduzidos segundo os custos definidos por b. O valor acumulado dessa propagação passa então a representar a distância ao fundo segundo a métrica induzida pela função estruturante.

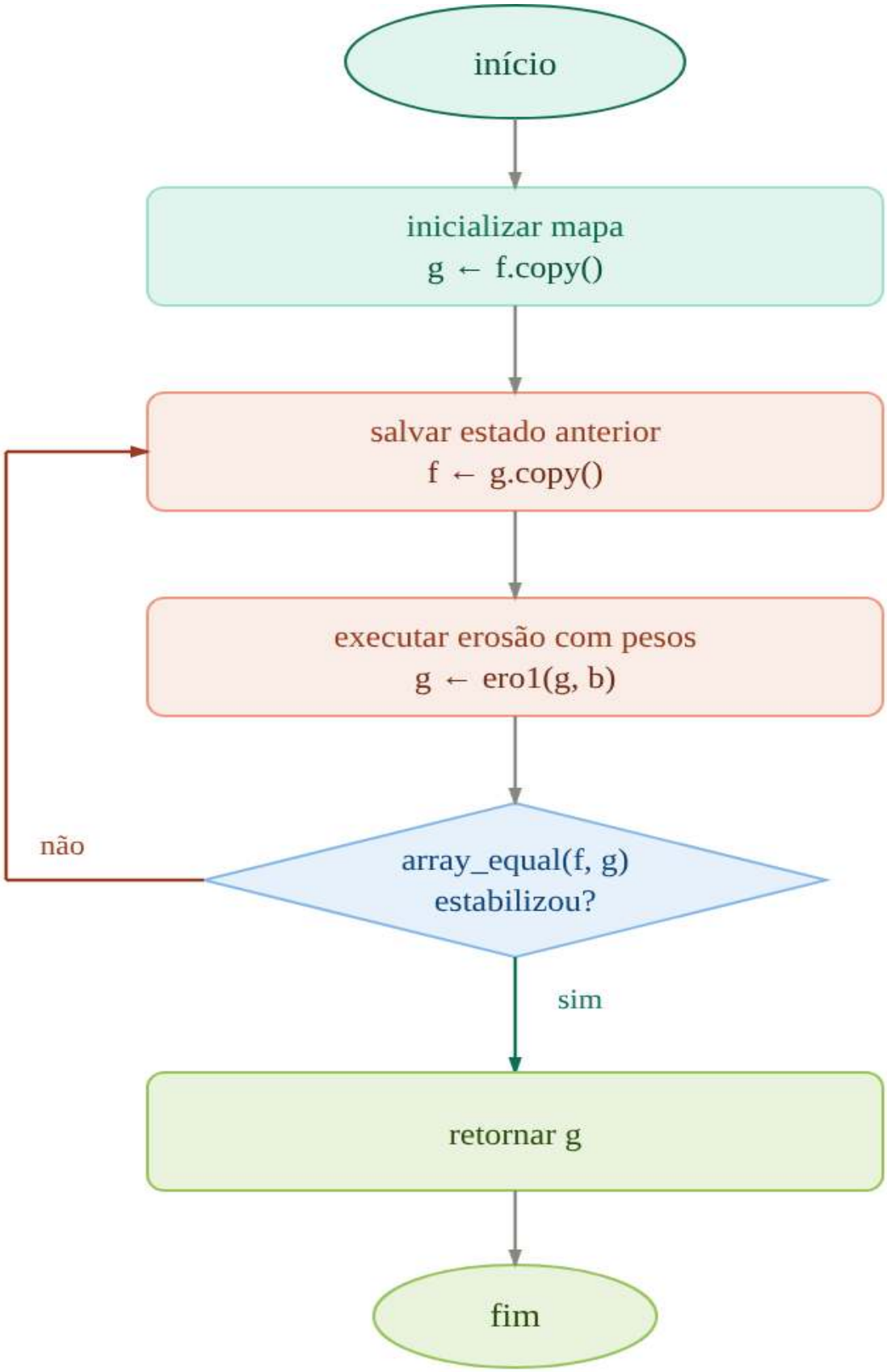
Essa interpretação é implementada em `mm::dist1()`, que acumula erosões sucessivas utilizando a operação `mm::ero1()`. Já `mm::dist()` delega o cálculo da distância Euclidiana ao operador otimizado do OpenCV `mm::dist(f)`, onde `f` é a imagem binária de entrada, distância L2 especifica a métrica Euclidiana (L_2) e `5` indica o uso de uma máscara 5×5 para aproximar a distância com elevada precisão.

A função `dist1` produz uma transformada de distância discreta cuja métrica é determinada pela geometria e pelos pesos da função estruturante utilizada. Por exemplo, utilizando uma função estruturante em cruz com custo unitário para os quatro vizinhos ortogonais, obtém-se a distância de ***Manhattan*** (L_1). Outras escolhas de vizinhança e pesos induzem métricas diferentes. Já `mm::dist()` calcula uma aproximação eficiente da distância Euclidiana (L_2).

Por exigir sucessivas erosões sobre toda a imagem, a abordagem `dist1` possui custo computacional significativamente maior que `mm::dist()`, sendo empregada neste livro principalmente para fins didáticos e para evidenciar a relação entre morfologia matemática e transformadas de distância.

A Figura 4.19 apresenta um simulador interativo da TD: é possível posicionar o cursor sobre diferentes pixels do objeto e observar, em tempo real, o valor da distância associado àquela posição, isto é, a distância até o pixel de fundo mais próximo. A Figura 4.20 apresenta um exemplo prático dessa execução em ambiente Python.

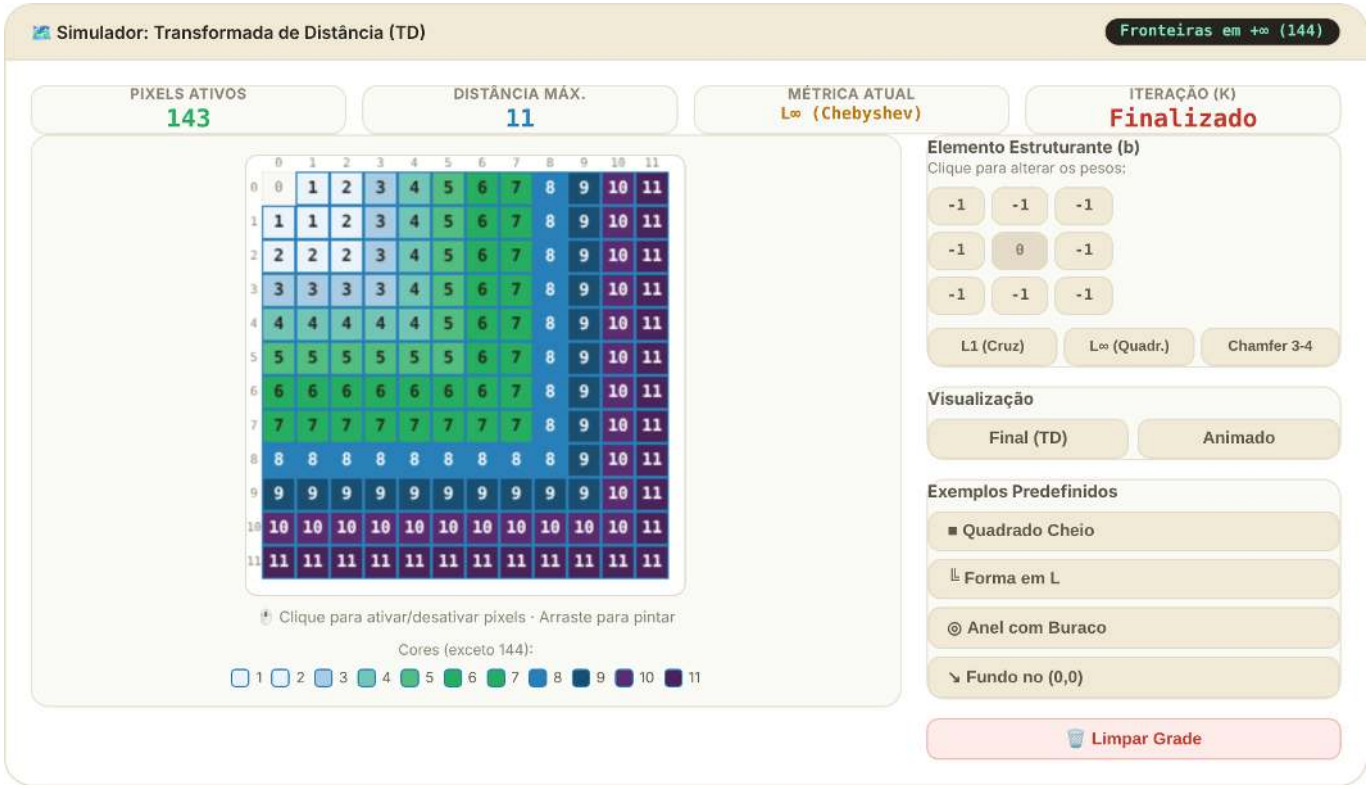
```
%%writefile tmp/fig_04_distancia_didatico.cpp
#define MM_OUT "tmp/fig_04_distancia_didatico.png"
//| label: fig-04-distancia-didatico
//| fig-cap: "Transformada de Distância em imagem binária 10×10. Esquerda: original
(*foreground* = 255). Centro: mm.dist1 iterativa (erosões com cruz). Direita: mm.dist (L2)."
//| echo: true
//| output: true

#include "morph.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#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, 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]
return 0;
}
```



Ponto Fixo Numérico: A distância emerge da propagação matemática do valor zero.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-alg-distancia2.html

Figura 4.18: Algoritmo da Transformada de Distância.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-04-distancia.html

Figura 4.19: Simulador interativo da Transformada de Distância (TD) iterativa via erosão em tons de cinza. Pixels fora da imagem assumem o valor máximo (144), propagando os custos a partir do fundo interno.

```
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)")
```

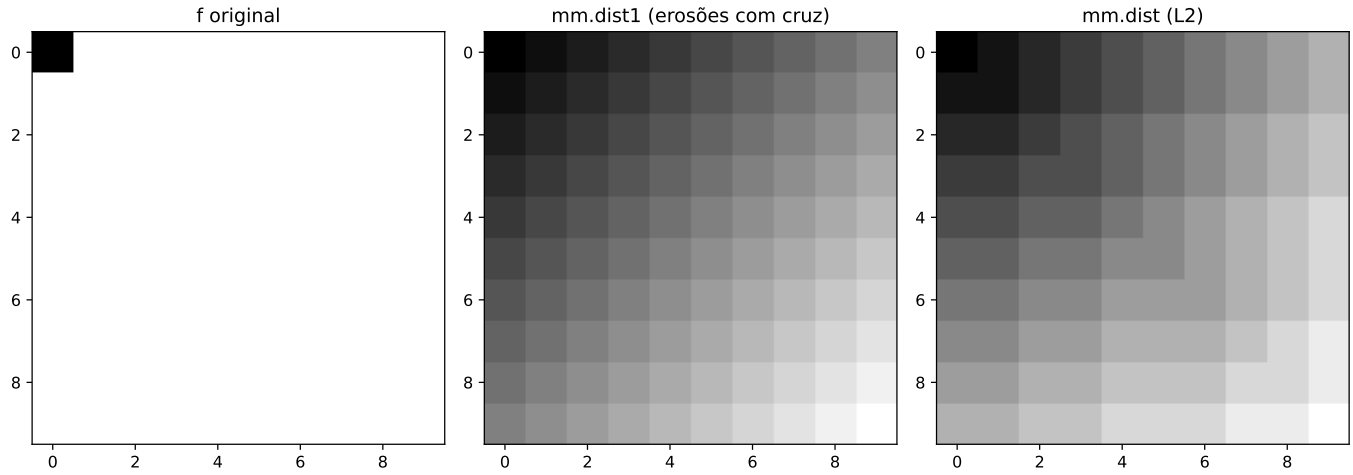


Figura 4.20: Transformada de Distância em imagem binária 10×10. Esquerda: original (*foreground* = 255). Centro: mm.dist1 iterativa (erosões com cruz). Direita: mm.dist (L2).

A anotação dos valores numéricos diretamente sobre os pixels permite verificar como `dist1` propaga as distâncias segundo a métrica induzida pela função estruturante utilizada. No caso do elemento cruz com custo unitário, os valores obtidos correspondem à distância de *Manhattan* (L_1). Embora `dist1` e `mm::dist` produzam valores numéricos distintos por adotarem métricas diferentes, ambas as transformadas preservam a estrutura topográfica dos objetos, fazendo com que seus máximos ocorram em regiões centrais semelhantes. Essa propriedade justifica o uso de `mm::dist` em aplicações práticas, devido à sua elevada eficiência computacional.

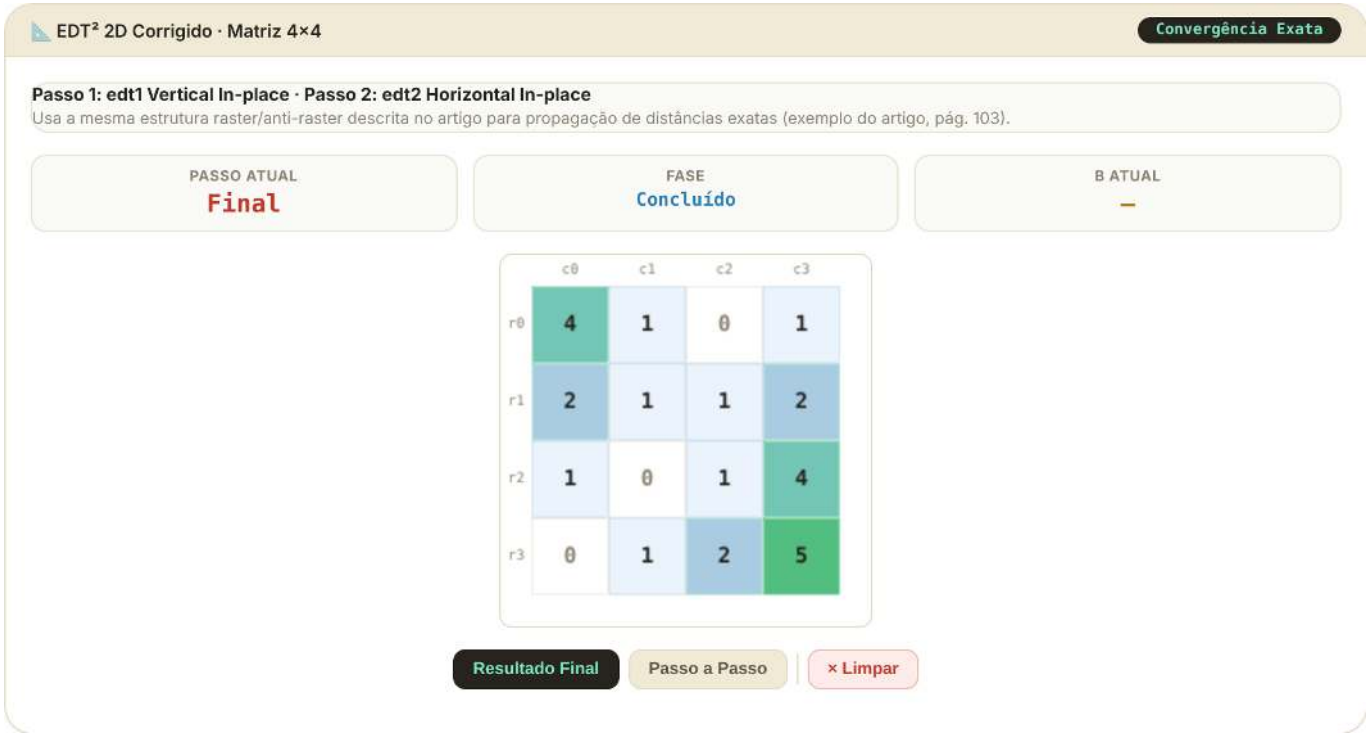
4.4.3 Transformada de Distância Euclidiana em quatro passos

O simulador da Figura 4.21 implementa o algoritmo da TDE de Lotufo (2001) em duas etapas. Na primeira etapa, a função `edt1` realiza uma transformação unidimensional vertical de forma sequencial (*in-place*), percorrendo cada coluna em *raster* (↓) e *anti-raster* (↑) para calcular as distâncias na direção vertical (dois passos: Sul e Norte). Na segunda etapa, a função `edt2` utiliza esse resultado como entrada e realiza uma propagação horizontal por filas: para cada linha da matriz, duas filas de prioridade, `Eq` e `Wq`, são inicializadas percorrendo os índices de coluna em sentidos opostos (`Eq` de `W-1` até `1`, `Wq` de `2` até `W`), de modo que cada pixel atualizado enfileira imediatamente seus vizinhos para reprocessamento dentro da mesma rodada (mais dois passos: Leste e Oeste). Esse mecanismo de fila permite aplicar erosões sucessivas com pesos ímpares crescentes (`b = 1, 3, 5, ...`, incrementado a cada iteração do laço externo) sem que a propagação fique presa em valores desatualizados, já que cada rodada resolve a cadeia de dependências horizontal por completo antes do próximo incremento de `b`. A convergência dessa propagação produz a Transformada de Distância Euclidiana em toda a matriz, combinando a informação vertical obtida em `edt1` com a propagação horizontal em fila realizada em `edt2`.

4.4.3.1 Transformada de Distância Geodésica

A transformada de distância geodésica associa a cada pixel a menor distância até um marcador, sob a restrição imposta por uma máscara. Dessa forma, a propagação ocorre exclusivamente pelos pixels permitidos, preservando a conectividade do domínio.

A Figura 4.22 ilustra esse processo em um labirinto: (a) a máscara `g`; (b) a distância geodésica `D1` calculada a partir da entrada; (c) a distância `D2` calculada a partir da saída; e (d) o caminho mínimo obtido a partir



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-04-tde.html

Figura 4.21: Simulador interativo 2D (4x4) com sincronização estrita de filas de propagação horizontal (b) para obter a convergência exata descrita no artigo.

dessas duas transformadas.

O caminho ótimo é determinado pela soma das distâncias (`D1 + D2`). Os pixels pertencentes à trajetória mínima são aqueles para os quais essa soma assume seu menor valor, definindo uma conexão entre entrada e saída com comprimento geodésico mínimo.

Esse princípio permite resolver labirintos sem a necessidade de explorar explicitamente todas as possibilidades de percurso. A solução emerge diretamente da propagação de distâncias em um domínio restrito. Essa abordagem é particularmente relevante em labirintos de elevada complexidade, como os construídos a partir de estruturas quasicristalinas e ciclos hamiltonianos descritos por Singh (2024). Em Zampirolli (2025), esse mesmo formalismo é empregado para resolver um labirinto complexo; a seguir, o método é ilustrado em uma versão simplificada do problema.

```
%%writefile tmp/fig_04_gdist_menor_caminho.cpp
#define MM_OUT "tmp/fig_04_gdist_menor_caminho.png"
//| 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

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

int main() {
    // 1 = corredor, 0 = parede
    // Convertendo o array numpy para mm::Image
    // O padrão é 0 (parede), então inicializamos com zeros e definimos os 1s
    mm::Image g(10, 10);
    // Preenchendo o labirinto linha por linha
```

```
    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; // valor máximo possível
    for (int i = 0; i < (int)S.data.size(); i++) {
        int val = S.data[i];
        if (val > 0)
            dmin = std::min(dmin, val);
    }

    // pixels pertencentes a um caminho ótimo
    mm::Image caminho(10, 10);
    for (int i = 0; i < (int)S.data.size(); i++)
        caminho.data[i] = (S.data[i] == dmin) ? 255 : 0;

    std::cout << "Distância geodésica mínima: " << dmin << "\n";

    mm::show(
        std::vector<mm::Image>{g, D1, D2, caminho},
        MM_OUT,
        std::vector<std::string>{
            "Labirinto",
            "Distância da Entrada",
            "Distância da Saída",
            "Menor Caminho\n(d=" + std::to_string(dmin) + ")"
        },
        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)")
```

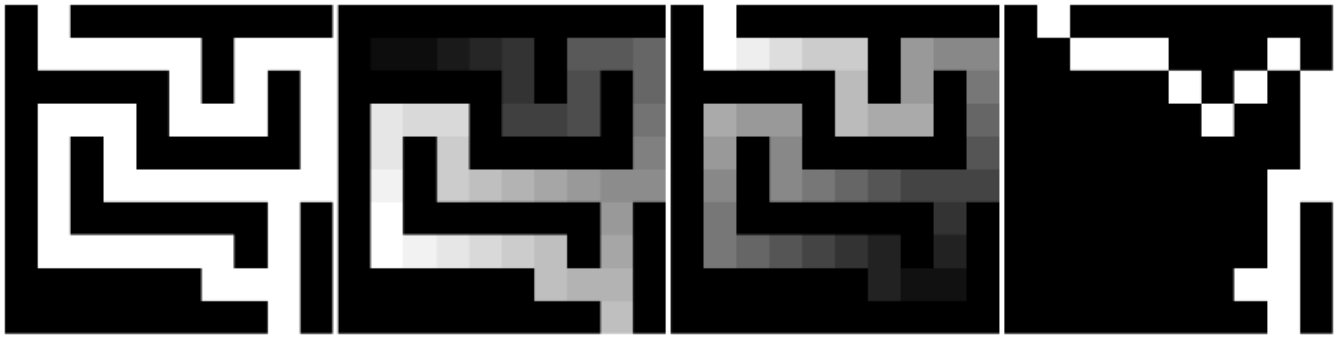


Figura 4.22: Menor caminho geodésico em um labirinto. As distâncias geodésicas são calculadas a partir da entrada e da saída usando mm.gdist. Os pixels cujo somatório das duas distâncias é igual à distância mínima entre os marcadores pertencem a um caminho ótimo.

4.4.4 Segmentação por *Watershed*

O algoritmo ***Watershed*** interpreta uma imagem em níveis de cinza como uma superfície topográfica, na qual valores elevados correspondem a montanhas e valores baixos correspondem a vales ou bacias de drenagem (*catchment basins*). No contexto da segmentação baseada em marcadores, os máximos da Transformada de Distância são frequentemente utilizados para identificar regiões internas aos objetos, fornecendo sementes confiáveis para o processo de inundação.

A segmentação é então realizada por uma simulação conceitual de inundação progressiva a partir desses marcadores. À medida que as bacias associadas a diferentes sementes se expandem, regiões vizinhas eventualmente entram em contato. Nesse instante são construídas barreiras virtuais, denominadas *watershed lines*, que passam a delimitar os objetos da cena. Esse mecanismo permite separar objetos adjacentes ou parcialmente sobrepostos, mesmo quando eles formam uma única componente conexa após a limiarização.

A implementação didática apresentada neste capítulo explora inicialmente o conceito de crescimento de regiões (*region growing*) confinado por uma máscara binária, conforme detalhado no algoritmo interativo da Figura 4.23.

i Versão didática versus implementação clássica

A função `mm::watershed0` não implementa o algoritmo *watershed* clássico. Seu objetivo é ilustrar, de forma simplificada, a propagação de marcadores por crescimento de regiões (*region growing*), permitindo visualizar como diferentes sementes competem pela ocupação do espaço disponível. O crescimento é delimitado por uma máscara binária de suporte e monitorado por um controle de estagnação, gerando um resultado semelhante a uma partição de Voronoi restrita à geometria dos objetos de entrada.
Já a função `mm::watershed` utiliza a implementação otimizada do OpenCV (`mm::watershed`), que realiza a inundação sobre uma superfície topográfica definida pela imagem de entrada. Nesse caso, a propagação dos marcadores é influenciada pelos valores dos pixels, fazendo com que as linhas de separação se formem naturalmente sobre as cristas do relevo.

A Figura 4.24 apresenta um simulador iterativo que ilustra a propagação dos marcadores pela região de interesse. Cada marcador atua como uma fonte de inundação que expande sua área de influência até encontrar regiões provenientes de outras sementes. No algoritmo do OpenCV, os pixels pertencentes às linhas divisoras são identificados pelo valor `-1`, representando as fronteiras entre bacias adjacentes.

Pipeline Morfológico do *Watershed*

O *watershed* baseado em marcadores normalmente integra um fluxo mais amplo de segmentação. Em imagens reais, etapas de pré-processamento são frequentemente necessárias para melhorar o contraste, reduzir ruídos e gerar marcadores confiáveis. Esse fluxo completo está resumido na Tabela 4.5.

Tabela 4.5: *Pipeline* completo do *watershed* baseado em marcadores para imagens reais.

Etapa	Operação	Finalidade
1	CLAHE + Suavização	Realce de contraste e redução de ruído
2	Limiarização	Separação inicial entre objeto e fundo
3	Abertura/Fechamento	Remoção de ruídos e pequenas imperfeições
4	Dilatação da Máscara	Identificação do Fundo Certo (*Sure Background*)
5	Transformada de Distância + Limiar	Identificação do Objeto Certo (*Sure Foreground*)
6	Região Incerta	Diferença entre Fundo Certo e Objeto Certo
7	`mm::watershed`	Propagação dos marcadores pela região incerta

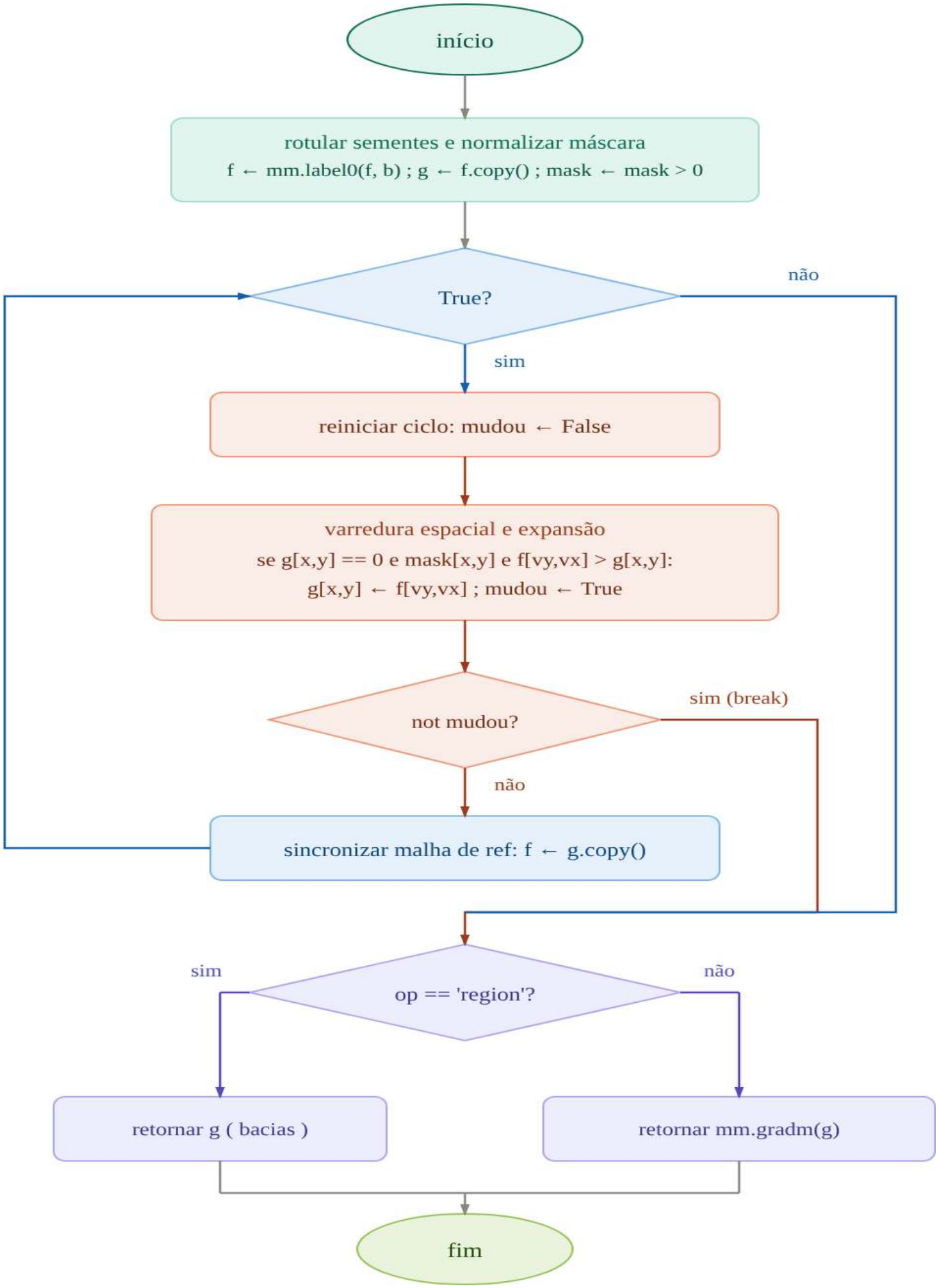
Para enfatizar exclusivamente os conceitos de Transformada de Distância, marcadores e inundação topográfica, o exemplo da Figura 4.25 utiliza uma imagem binária sintética e adota um fluxo simplificado, resumido na Tabela 4.6.

Tabela 4.6: *Pipeline* simplificado utilizado no exemplo didático da Figura 4.25.

Etapa	Operação	Finalidade
1	Transformada de Distância	Construção da superfície topográfica
2	Limiar da TD	Extração dos marcadores (*Sure Foreground*)

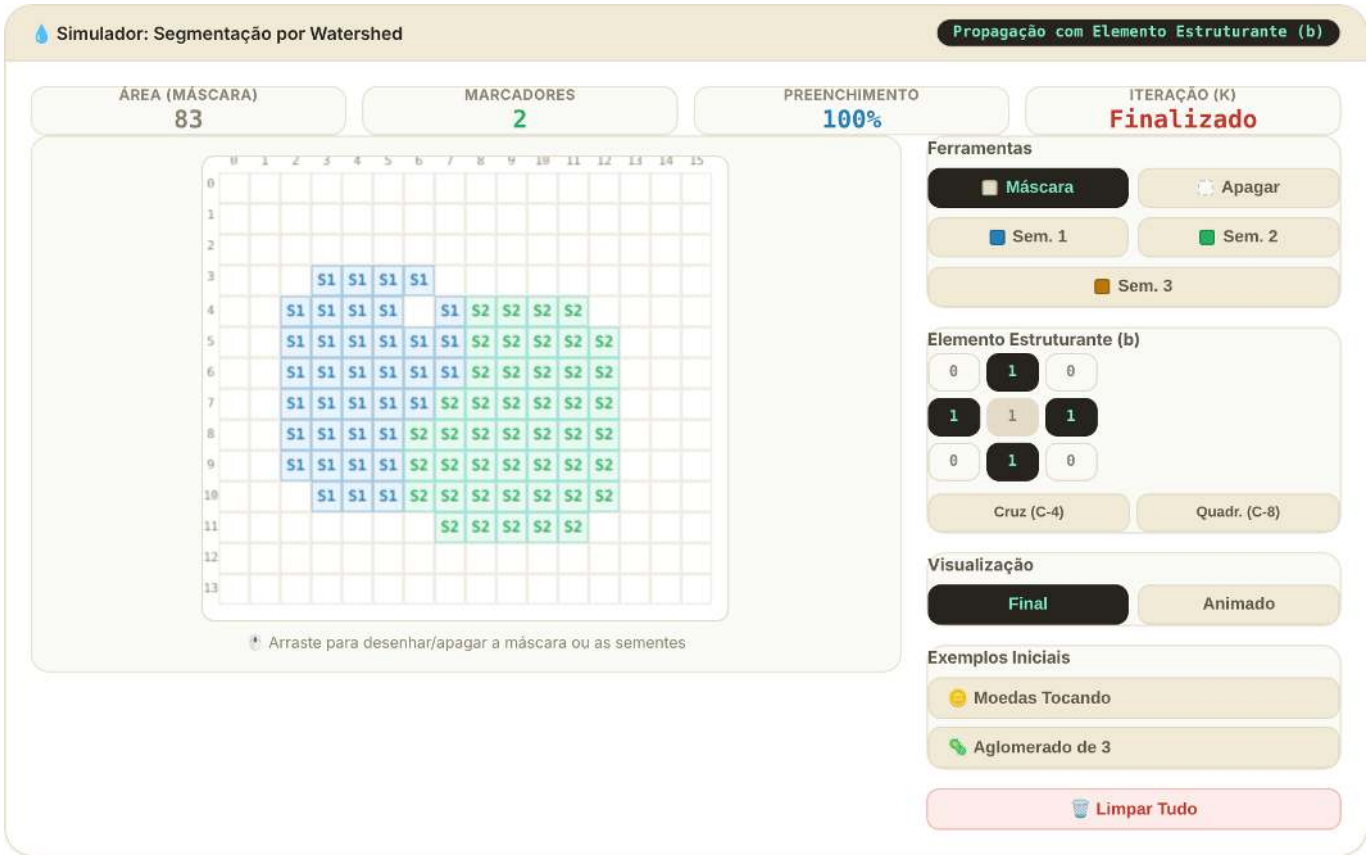
Etapa	Operação	Finalidade
3	Dilatação	Determinação do Fundo Certo (*Sure Background*)
4	Região Incerta	Diferença entre fundo e marcadores
5	`mm::watershed`	Propagação dos marcadores e geração das fronteiras

```
%%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. Imagem sintética e Transformada de Distância
    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. Marcadores (picos da distância)
    mm::Image m(20, 20);
    int maxDist = 0;
    for (int y = 0; y < dist.h; y++) {
        for (int x = 0; x < dist.w; x++) {
            if ((int)dist.at(y, x) > maxDist) maxDist = dist.at(y, x);
        }
    }
    for (int y = 0; y < dist.h; y++) {
        for (int x = 0; x < dist.w; x++) {
            m.at(y, x) = ((int)dist.at(y, x) > 0.8 * maxDist) ? 255 : 0;
        }
    }

    // 3. Execução do Watershed0 condicional (m=marcadores primeiro, 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, f_sint, "region");
    //w_line = mm::watershedB(m, f_sint, "line");

    w_reg  = mm::watershed(m, f_sint, "region");
    w_line = mm::watershed(m, f_sint, "line");

    // 4. Exibição dos resultados
    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");
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-alg-watershed2.html

Figura 4.23: Algoritmo Didático de *Watershed* por Crescimento de Regiões Limitado por Máscara.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-04-watershed.html

Figura 4.24: Simulador interativo do Algoritmo *Watershed* por propagação morfológica. Desenhe a máscara, posicione os marcadores e ajuste o Elemento Estruturante para observar a inundação. Quando as bacias se encontram simultaneamente, o empate é resolvido assumindo uma das regiões de forma aleatória.

```
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]
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)")
```

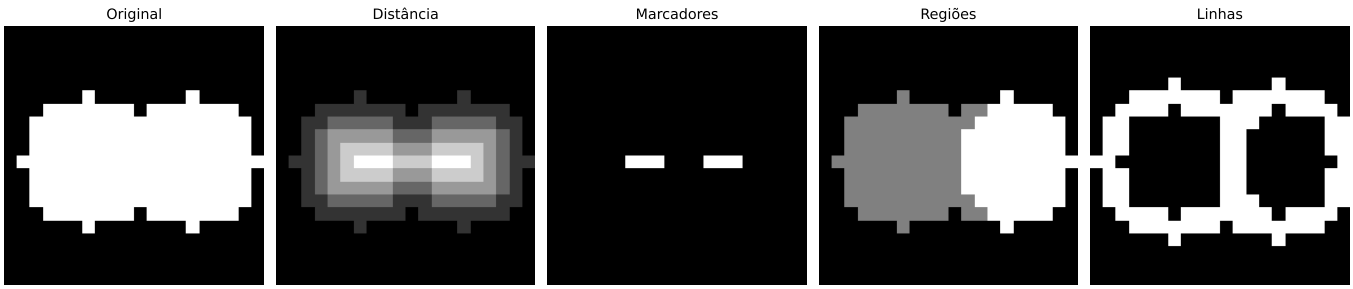


Figura 4.25: *Pipeline watershed* delimitado por máscara em imagem binária 20×20.

Aplicação em moedas sobrepostas:

```
%%writefile tmp/fig_04_watershed_moedas.cpp
#define MM_OUT "tmp/fig_04_watershed_moedas.png"
#include "morph.hpp"
#include <opencv2/opencv.hpp>
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <numeric>
#include <set>
#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. Simular moedas sobrepostas/conectadas (usando a máscara binária base)
    mm::Image img_sobrepostas = mm::dil(img_final, mm::sebox(40));

    // 2. Abertura morfológica para limpar ruídos
    mm::Image opening = mm::open(img_sobrepostas, mm::sebox(2));

    // 3. Transformada de Distância
    mm::Image dist = mm::dist(opening);

    // Encontrar max da distância para normalização
    int max_dist = 0;
    for (int i = 0; i < dist.data.size(); i++) {
        max_dist = std::max(max_dist, (int)dist.data[i]);
    }

    mm::Image dist_vis(dist.h, dist.w);
    for (int i = 0; i < dist.data.size(); i++) {
        if (max_dist > 0) {
            dist_vis.data[i] = (unsigned char)(255 * dist.data[i] / max_dist);
        } else {
            dist_vis.data[i] = dist.data[i];
        }
    }

    // 4. Picos seguros (Marcadores das moedas)
    mm::Image picos(dist.h, dist.w);
    for (int i = 0; i < dist.data.size(); i++) {
        if (dist.data[i] > 0.5 * max_dist) {
            picos.data[i] = 255;
        } else {
            picos.data[i] = 0;
        }
    }

    // 5. Execução do Watershed (Ajustado para usar a nova assinatura)
```

```
    // Passamos 'opening' direto em mask, pois ela delimita o escopo de expansão das moedas
    mm::Image ws_region = mm::watershed(picos, opening, "region");
    mm::Image ws_line = mm::watershed(picos, opening, "line");

    // 6. Contagem de objetos (Ignora fundo 0)
    std::set<int> unique_labels;
    for (int i = 0; i < ws_region.data.size(); i++) {
        if (ws_region.data[i] > 0) {
            unique_labels.insert((int)ws_region.data[i]);
        }
    }
    std::vector<int> labels(unique_labels.begin(), unique_labels.end());
    std::cout << "Objetos detectados: " << labels.size() << "\n";

    // 7. Anotação Final
    cv::Mat img_base_cv(img_base.h, img_base.w, CV_8UC1, img_base.data.data());
    cv::Mat img_annotated;
    cv::cvtColor(img_base_cv, img_annotated, cv::COLOR_GRAY2BGR);

    // Criar máscara para cada rótulo e anotar
    cv::RNG rng(12345);
    for (size_t idx = 0; idx < labels.size(); idx++) {
        int label_id = labels[idx];

        // Criar máscara para este rótulo
        mm::Image mask_reg(ws_region.h, ws_region.w);
        long sum = 0;
        for (int i = 0; i < ws_region.data.size(); i++) {
            if ((int)ws_region.data[i] == label_id) {
                mask_reg.data[i] = 255;
                sum++;
            }
        }

        if (sum < 500) continue;

        // Encontrar centroide
        int sum_x = 0, sum_y = 0, 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) == 255) {
                    sum_x += x;
                    sum_y += y;
                    count++;
                }
            }
        }
        int cx = sum_x / count;
        int cy = sum_y / count;

        // Anotar com número
        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);
    }

    // Desenha as linhas de separação em vermelho
    mm::Kernel k_ones(11, 11);
    for (int i = 0; i < k_ones.vals.size(); i++) k_ones.vals[i] = 1.0;

    mm::Image mask_ann = mm::dil(ws_line, mm::SE::box(11));
```

```
    // Aplicar cor vermelha onde mask_ann > 0
    for (int y = 0; y < img_annotated.rows; y++) {
        for (int x = 0; x < img_annotated.cols; x++) {
            if (mask_ann.at(y, x) > 0) {
                cv::Vec3b& pixel = img_annotated.at<cv::Vec3b>(y, x);
                pixel[0] = 0;  // B
                pixel[1] = 0;  // G
                pixel[2] = 255; // R
            }
        }
    }

    // Converter de volta para mm::Image para exibição
    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());

    // Exibição
    mm::show(
        std::vector<mm::Image>{img_base, img_sobrepostas, dist_vis, picos, ws_region,
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)")
```

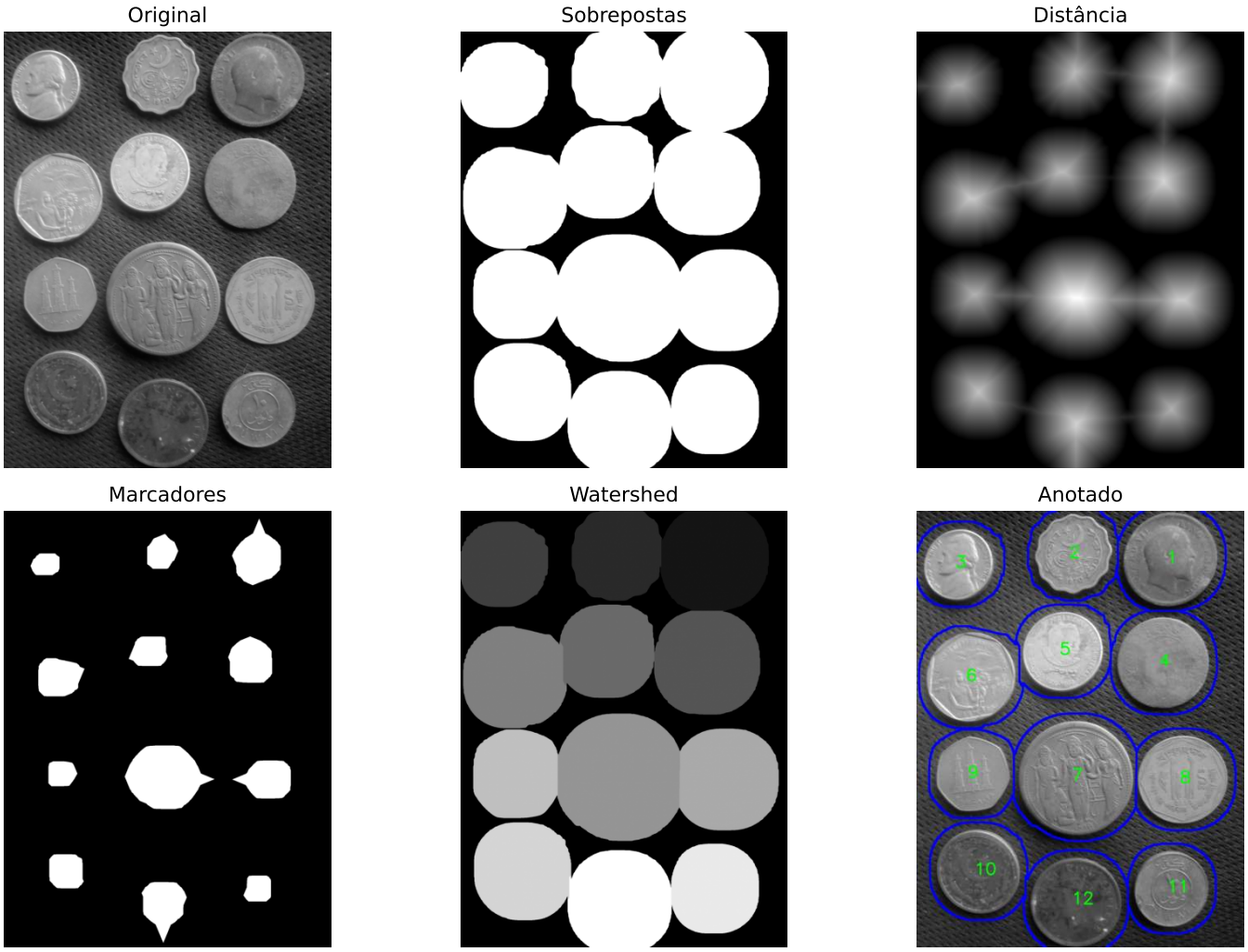


Figura 4.26: *Pipeline Watershed* para separação de moedas sobrepostas: da máscara binária dilatada até os contornos finais anotados sobre a imagem original.

4.5 Extração de Componentes e Descritores de Forma

Após a segmentação e o refinamento morfológico, o próximo passo consiste em identificar individualmente cada objeto presente na imagem e extrair suas propriedades geométricas. Essa etapa é fundamental para tarefas de medição, classificação e reconhecimento de padrões.

Um **componente conexo** é um conjunto maximal de pixels pertencentes ao objeto que permanecem mutuamente conectados segundo uma relação de conectividade previamente definida (4- ou 8-conectividade). Após a rotulação, cada componente recebe um identificador único, permitindo que suas características sejam analisadas individualmente.

O OpenCV oferece duas abordagens complementares para essa análise, resumidas na Tabela 4.7.

Tabela 4.7: Comparação entre as abordagens baseadas em componentes conexos e em contornos.

	`connectedComponentsWithStats`	`findContours`
Retorna	rótulo por pixel e estatísticas por componente	sequência de pontos que descreve a borda
Descritores diretos	área, *bounding box* e centróide	perímetro, forma e hierarquia
Objetos em contato	tende a fundir regiões conectadas	tende a produzir um único contorno externo
Uso típico	contagem, filtragem e rotulação	análise geométrica e descritores de forma

4.5.1 Rotulação e Estatísticas de Componentes

`mm::label0` atribui um rótulo a cada componente conexo. A partir da imagem rotulada obtêm-se, por varredura, a **área** (contagem de pixels) e a **caixa delimitadora** de cada objeto (Figura 4.27). O `connectedComponentsWithStats` do OpenCV, que devolve essas estatísticas prontas, e as anotações coloridas sobre a imagem ficam na trilha Python.

```
%%writefile tmp/fig_04_componentes.cpp
#define MM_OUT "tmp/fig_04_componentes.png"
#include "morph.hpp"
#include <opencv2/opencv.hpp>
#include <cstdio>
#include <vector>
#include <string>
#include <cstring>
#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]

    // Variáveis fornecidas: img_coins_gray, img_final

    // 1. Rotulagem e estatísticas
    cv::Mat mat_final(img_final.h, img_final.w, img_final.channels == 1 ? CV_8UC1 : CV_8UC3,
img_final.data.data());
    cv::Mat labels, stats, centroids;
    int n = cv::connectedComponentsWithStats(mat_final, labels, stats, centroids, 8);

    // 2. Coloração dos componentes - paleta FIXA
    cv::Vec3b paleta[] = {
```

```
        cv::Vec3b(0, 0, 0), cv::Vec3b(224, 82, 193), cv::Vec3b(233, 155, 73), cv::Vec3b(190,
247, 244),
        cv::Vec3b(103, 94, 179), cv::Vec3b(51, 154, 59), cv::Vec3b(137, 96, 232),
cv::Vec3b(250, 243, 205),
        cv::Vec3b(100, 141, 208), cv::Vec3b(228, 187, 163), cv::Vec3b(202, 108, 214),
cv::Vec3b(131, 105, 104),
        cv::Vec3b(86, 153, 81)
    };
    const int paleta_size = 13;

    cv::Mat img_colored(img_coins_gray.h, img_coins_gray.w, CV_8UC3, cv::Scalar(0, 0, 0));

    for (int y = 0; y < img_final.h; y++) {
        for (int x = 0; x < img_final.w; x++) {
            int label = labels.at<int>(y, x);
            cv::Vec3b cor;
            if (label == 0) {
                cor = cv::Vec3b(0, 0, 0);
            } else {
                int idx = label % paleta_size;
                if (idx == 0) idx = paleta_size - 1; // evita usar preto para componentes
                cor = paleta[idx];
            }
            img_colored.at<cv::Vec3b>(y, x) = cor;
        }
    }

    cv::Mat img_annotated = img_colored.clone();

    // 3. Tabela de descritores
    std::printf("Componentes detectados (excluindo fundo): %d\n", n - 1);
    std::printf("\n%4s %8s %6s %6s %6s %6s\n", "ID", "Área", "cx", "cy", "w", "h");
    std::printf("%s\n", "-----------------------------------------");

    for (int i = 1; i < n; i++) {
        int area = stats.at<int>(i, cv::CC_STAT_AREA);
        double cx_d = centroids.at<double>(i, 0);
        double cy_d = centroids.at<double>(i, 1);
        int cx = (int)cx_d;
        int cy = (int)cy_d;
        int w = stats.at<int>(i, cv::CC_STAT_WIDTH);
        int h = stats.at<int>(i, cv::CC_STAT_HEIGHT);

        std::printf("%4d %8d %6d %6d %6d %6d\n", i, area, cx, cy, w, h);

        char texto[64];
        std::snprintf(texto, sizeof(texto), "%d: %d", i, area);

        // Contorno preto e depois vermelho para melhor legibilidade
        cv::putText(img_annotated, texto, cv::Point(cx - 150, cy + 18),
                    cv::FONT_HERSHEY_SIMPLEX, 2.0, cv::Scalar(0, 0, 0), 10, cv::LINE_AA);
        cv::putText(img_annotated, texto, cv::Point(cx - 150, cy + 18),
                    cv::FONT_HERSHEY_SIMPLEX, 2.0, cv::Scalar(255, 0, 0), 5, cv::LINE_AA);
    }

    // Conversão do resultado OpenCV para mm::Image
    mm::Image img_colored_mm(img_colored.rows, img_colored.cols, 3);
    std::memcpy(img_colored_mm.data.data(), img_colored.data, img_colored_mm.data.size());

    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());

    mm::show(
        std::vector<mm::Image>{img_coins_gray, img_final, img_colored_mm, img_annotated_mm},
        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)")
```

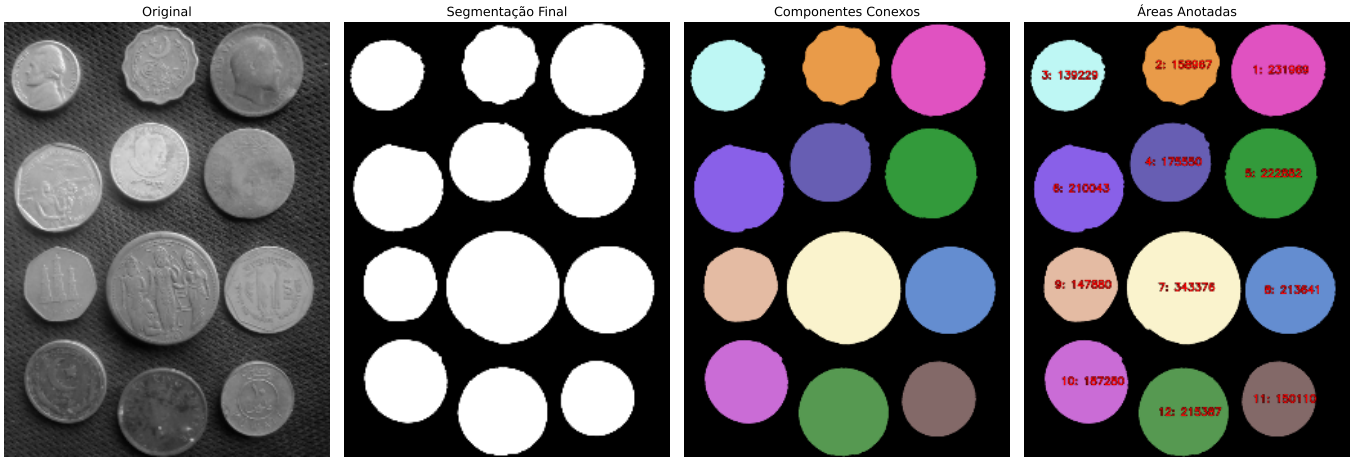


Figura 4.27: Componentes conexos extraídos após o *pipeline* CLAHE → Otsu → limpeza morfológica. Cada objeto é colorido com cor distinta e anotado com sua área em pixels.

4.5.2 Descritores de Forma

Descritores baseados em **contorno vetorial** — perímetro (`arcLength`), área do polígono (`contourArea`), circularidade, momentos e aproximação poligonal (`approxPolyDP`) — dependem de rastreamento de fronteira (`findContours`), ausente na `morph.hpp`. Ficam só na trilha Python. Na trilha C++, o **gradiente morfológico** (`mm::gradm`) evidencia o contorno de cada objeto (Figura 4.28).

```
%%writefile tmp/fig_04_contornos.cpp
#define MM_OUT "tmp/fig_04_contornos.png"
//| 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

#include "morph.hpp"
#include <opencv2/opencv.hpp>
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <cstring>
#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]

    // img_coins_gray and img_final are provided as already-valid mm::Image variables

    // Bridge mm::Image to cv::Mat for OpenCV operations
    cv::Mat 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(final_mat, contornos, cv::RETR_EXTERNAL, cv::CHAIN_APPROX_SIMPLE);

    // Convert grayscale to BGR for visualization
    cv::Mat gray_mat(img_coins_gray.h, img_coins_gray.w, CV_8UC1, img_coins_gray.data.data());
    cv::Mat contornos_mat;
    cv::cvtColor(gray_mat, contornos_mat, cv::COLOR_GRAY2BGR);
    cv::Mat circulares_mat = contornos_mat.clone();

    std::cout << "Contornos detectados: " << contornos.size() << "\n";
    std::cout << "\n" << "  ID" << " " << "    Área" << " " << "  Perímetro" << " " << "
Circularidade" << "\n";
    std::cout << "-----------------------------------------\n";

    int i = 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 << i << " " << area << " " << perim << " " << circ << "\n";

        std::vector<std::vector<cv::Point>> single_cnt = {cnt};
        cv::drawContours(contornos_mat, single_cnt, -1, cv::Scalar(0, 255, 0), 3);
        cv::drawContours(circulares_mat, single_cnt, -1, cv::Scalar(0, 255, 0), 3);

        // Draw text with outline
        std::string circ_text = std::to_string(static_cast<int>(circ * 100) / 100.0);
        circ_text = circ_text.substr(0, circ_text.find('.') + 3);  // Format to 2 decimal
places

        for (const auto& [color, thickness] : std::vector<std::pair<cv::Scalar, int>>{
            {cv::Scalar(0, 0, 0), 8}, {cv::Scalar(255, 0, 0), 3}}) {
            cv::putText(circulares_mat, circ_text, cv::Point(cx - 80, cy + 15),
                        cv::FONT_HERSHEY_SIMPLEX, 3.6, color, thickness, cv::LINE_AA);
        }
        i++;
    }

    // Convert results back to mm::Image
    mm::Image img_contornos(contornos_mat.rows, contornos_mat.cols, 3);
    std::memcpy(img_contornos.data.data(), contornos_mat.data, img_contornos.data.size());

    mm::Image img_circulares(circulares_mat.rows, circulares_mat.cols, 3);
    std::memcpy(img_circulares.data.data(), circulares_mat.data, img_circulares.data.size());

    mm::show(
        std::vector<mm::Image>{img_final, img_contornos, img_circulares},
        MM_OUT,
        std::vector<std::string>{"Segmentação Final", "Contornos", "Circularidade"},
```

```
        3
    );


// [pdi:panel-io] auto-generated - do not edit by hand
std::filesystem::create_directories("tmp");
mm::write(img_final, "tmp/fig_04_contornos_0.png");
mm::write(img_contornos, "tmp/fig_04_contornos_1.png");
mm::write(img_circulares, "tmp/fig_04_contornos_2.png");
// [pdi:panel-io:end]
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     Área   Perímetro   Circularidade
-----------------------------------------
1 214626 1769.6 0.861278
2 149480 1465.11 0.875093
3 186570 1654.93 0.856037
4 147252 1458.62 0.869733
5 212886 1756.29 0.867293
6 342424 2232.54 0.863326
7 209282 1767.7 0.841637
8 222108 1809.68 0.852256
9 174854 1616.4 0.840983
10 138602 1471.53 0.804339
11 158306 1538.7 0.840227
12 231181 1847.38 0.851234
[1] Segmentação Final
[2] Contornos
[3] Circularidade
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_04_contornos_0.png"),
            mm.read("tmp/fig_04_contornos_1.png"),
            mm.read("tmp/fig_04_contornos_2.png"),
        ],
        titles=[
            'Segmentação Final',
            'Contornos',
            'Circularidade',
        ],
        cols=3,
        figsize=(18, 6),
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + " tmp/fig_04_contornos_0.png
    (ver a versao Python)")
```

Figura 4.28: Contornos extraídos com *findContours*. Cada moeda é anotada com sua circularidade — valores próximos de 1 confirmam forma circular.

4.5.3 Conexão com Detecção de Objetos Moderna

Os descritores extraídos nas seções anteriores — especialmente *bounding boxes*, centróides, áreas e medidas de forma — estabelecem uma ponte natural entre a segmentação morfológica clássica e os sistemas modernos de detecção de objetos. Embora as técnicas estudadas neste capítulo utilizem operações sobre pixels e regiões segmentadas, muitas das representações produzidas são diretamente compatíveis com os formatos empregados em modelos contemporâneos de visão computacional.

Detectores baseados em aprendizado profundo, como a família YOLO (*You Only Look Once*) (REDMON, 2016), operam diretamente sobre imagens coloridas e produzem, para cada objeto detectado, uma *bounding box* descrita pelo centro (cx, cy) e pelas dimensões (w, h), além de uma classe e uma pontuação de confiança. Essa representação compartilha a mesma estrutura geométrica básica obtida por `connectedComponentsWithStats`, embora seja produzida por um modelo aprendido e não por segmentação explícita.

A Figura 4.29 ilustra como as *bounding boxes* obtidas por morfologia podem ser exportadas no formato YOLO para compor conjuntos de dados utilizados no treinamento ou na avaliação de detectores.

```
%%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 <opencv2/opencv.hpp>
#include <cstdio>
#include <string>
#include <vector>
#include <fstream>
#include "morph.hpp"
#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)

    printf("%4s %8s %8s %8s %8s  ← formato YOLO\n", "cls", "cx_n", "cy_n", "w_n", "h_n");
    printf("-----------------------------------------------------\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  = (double)w / W_img;
        double h_n  = (double)h / H_img;

        char linha[128];
        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++) {
        f << yolo_linhas[i];
        if (i + 1 < yolo_linhas.size()) f << "\n";
    }
    f.close();
    printf("\nAnotação salva em moedas.txt\n");

    // Converter img_bbox_cv de volta para 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(
        {img_coins_gray, img_final, img_bbox},
        MM_OUT,
```

```
        {"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)")
```

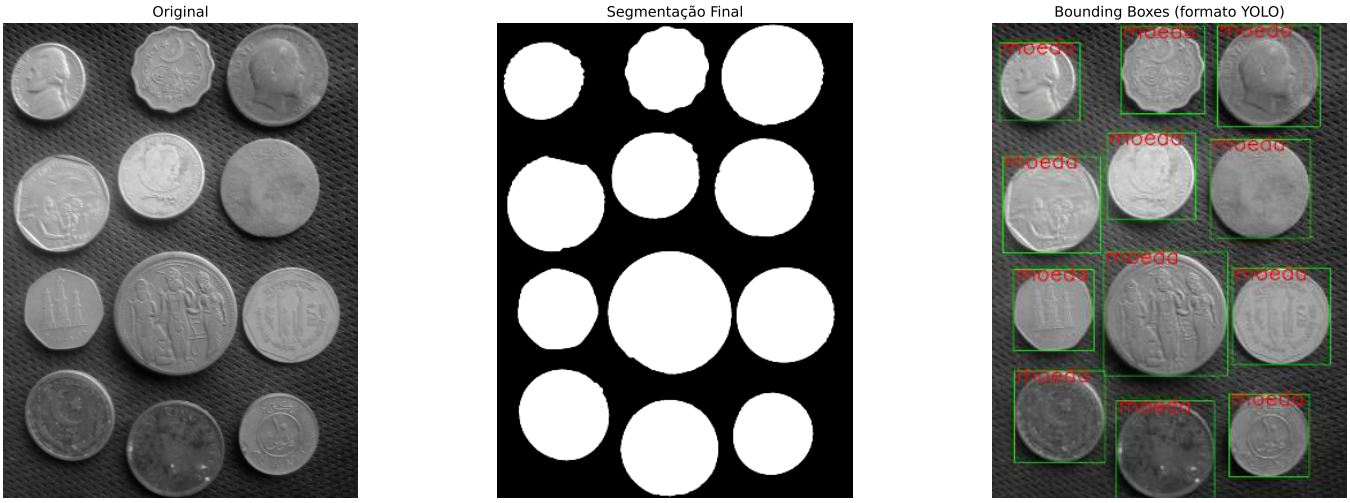


Figura 4.29: *Bounding boxes* derivadas dos componentes conexos sobrepostas à imagem original. As anotações são exportadas no formato YOLO (*classe cx cy w h*), com coordenadas normalizadas para o intervalo [0,1].

O formato YOLO armazena cada objeto em uma linha contendo cinco campos:

```
classe cx cy w h
```

onde (cx, cy) representa o centro da *bounding box* e (w, h) suas dimensões. Todos os valores geométricos são normalizados para o intervalo $[0, 1]$ em relação à largura e à altura da imagem. A **classe** é um identificador inteiro associado a uma categoria definida pelo conjunto de dados (por exemplo, `0 → moeda`). Quando há múltiplas categorias — como moeda de ouro (`0`), moeda de prata (`1`) e disco plástico (`2`) — basta atribuir o identificador correspondente a cada objeto antes da exportação, mantendo exatamente o mesmo formato de anotação. No exemplo anterior, essas informações foram armazenadas no arquivo `moedas.txt`.

O fluxo apresentado neste capítulo — segmentação → rotulação → extração de *bounding boxes* — corresponde conceitualmente à etapa de **anotação** (*labeling*) empregada na construção de conjuntos de treinamento para detectores modernos. Ferramentas especializadas, como Label Studio e Roboflow, automatizam esse processo em imagens complexas, mas a lógica fundamental permanece a mesma: associar a cada objeto uma região de interesse e uma classe. Em cenários controlados, com fundo uniforme e objetos bem separados, técnicas morfológicas podem inclusive gerar anotações automaticamente ou servir como ponto de partida para a rotulação manual, reduzindo significativamente o esforço de construção do conjunto de dados. Em aplicações reais mais complexas, entretanto, a validação humana continua sendo necessária para garantir a qualidade das anotações.

i Avaliação: IoU (*Intersection over Union*)

Uma forma simples de avaliar a qualidade de uma segmentação consiste em compará-la com uma máscara de referência (*ground truth*). A métrica mais utilizada para essa finalidade é a **IoU** (*Intersection over Union*):

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} \tag{4.16}$$

onde A representa a segmentação produzida pelo algoritmo e B a segmentação de referência.
O valor da IoU varia entre 0 e 1. Quanto maior o valor, maior a sobreposição entre as máscaras. Uma IoU igual a 1 indica correspondência perfeita entre a segmentação obtida e a referência.
A mesma métrica também é amplamente utilizada em detecção de objetos, sendo aplicada às *bounding boxes* previstas e anotadas. Nessa área, valores de IoU superiores a 0,5 são frequentemente adotados como critério mínimo para considerar uma detecção correta.

Além da Morfologia

As técnicas estudadas neste capítulo segmentam objetos explorando conectividade espacial, operações morfológicas e relevo topográfico. Existem, entretanto, abordagens alternativas baseadas em agrupamento de características, como o algoritmo *k-means*, modelos de mistura Gaussiana (GMM) e métodos mais recentes baseados em aprendizado profundo. Essas técnicas serão retomadas na Parte II do livro, dedicada à Visão Computacional.

4.6 Resumo

Neste capítulo foram apresentadas as principais técnicas de segmentação e morfologia matemática, concluindo o estudo do PDI no domínio espacial.

- **Pré-processamento e limiarização:** A combinação entre equalização adaptativa CLAHE e o método de Otsu mostrou-se eficaz para induzir separação bimodal no histograma e simplificar a binarização de imagens com iluminação não uniforme.
- **Erosão e dilatação:** Operadores morfológicos fundamentais baseados na busca por mínimos e máximos locais em uma vizinhança definida pelo elemento estruturante B. São operadores duais pelo complemento e foram implementados por meio das funções `mm::ero` e `mm::dil`.
- **Abertura e fechamento:** Composições de erosão e dilatação que permitem remover ruídos, suavizar contornos e preencher pequenas lacunas, preservando a estrutura global dos objetos.
- **Reconstrução morfológica:** Processo geodésico iterativo que propaga um marcador dentro dos limites impostos por uma máscara, constituindo a base de operadores como `mm::clohole` e `mm::edgeoff`.
- **Pipeline de limpeza binária:** Fluxo consolidado composto por CLAHE → Otsu → abertura → `mm::clohole` → abertura restrita → `mm::edgeoff`, produzindo máscaras adequadas para análise quantitativa.
- **Morfologia em tons de cinza:** Extensão algébrica baseada em mínimos e máximos ponderados, viabilizando operadores como gradiente morfológico e filtros *top-hat* para realce de estruturas locais.
- **Transformada de Distância e Watershed:** A Transformada de Distância permitiu gerar marcadores automáticos para o algoritmo *watershed*, possibilitando a separação de objetos adjacentes ou parcialmente sobrepostos.
- **Componentes conexos e descritores:** A rotulação de regiões (`mm::label0`) e a extração de contornos (extração de contornos) permitiram calcular descritores geométricos como área, centróide, perímetro, circularidade e *bounding boxes*.
- **Conexão com Visão Computacional Moderna:** As *bounding boxes* extraídas por morfologia foram exportadas no formato YOLO, evidenciando a ligação entre técnicas clássicas de segmentação e sistemas modernos de detecção de objetos.

O Capítulo 5 introduzirá técnicas de processamento no **domínio da frequência**, abordando a **Transformada de Fourier**, filtragem espectral e os fundamentos da **compressão de imagens**, incluindo DCT, JPEG e *wavelets*.

4.7 🤖 Uso do Gemini Notebook como Tutor Complementar

Nesta edição, o uso do **Gemini Notebook** é incentivado como ferramenta complementar de aprendizagem. Baseado em inteligência artificial, o sistema utiliza exclusivamente os documentos fornecidos pelo autor como fonte de conhecimento, produzindo respostas alinhadas ao conteúdo e à abordagem adotada ao longo deste capítulo.

🎓 Estude com o Tutor Inteligente

🚀 ACESSAR Gemini Notebook: CAPÍTULO 04

🌐 **Idioma e Linguagem de Programação**

O projeto deste capítulo no Gemini Notebook foi construído apenas com o texto em **português** e os exemplos de código em **Python**. Se você está estudando pela edição em inglês ou francês, ou acompanhando a trilha em C++, as respostas do tutor podem não corresponder exatamente à versão que você está lendo.

⚠️ **Aviso sobre Conteúdo Gerado por IA**

Embora seja uma ferramenta valiosa de apoio aos estudos, o Gemini Notebook pode eventualmente produzir respostas incompletas, imprecisas ou incorretas. Recomenda-se validar as informações consultando o material do capítulo, livros, artigos científicos e outras fontes acadêmicas confiáveis. Sempre que possível, execute e experimente os exemplos práticos apresentados ao longo do texto para consolidar a compreensão dos conceitos.

4.8 Lista de Exercícios

1. **(10%)** Implemente manualmente o critério de Otsu sem utilizar `mm::threshold`. Calcule a variância interclasses $\sigma_B^2(T)$ para todos os limiares $T \in [0, 255]$ usando `mm::hist`, identifique o limiar ótimo T^* e compare o resultado com o valor obtido pelo OpenCV. Plote σ_B^2 em função de T e destaque o ponto de máximo.

2. **(15%)** Aplique limiarização adaptativa com blocos de tamanho 11, 31 e 51 a uma imagem contendo iluminação não uniforme. Compare os resultados com a limiarização global de Otsu e discuta as vantagens e limitações de cada abordagem.

3. **(15%)** Execute o *pipeline watershed* da imagem de moedas variando o limiar aplicado à Transformada de Distância (0.3, 0.5 e 0.7 vezes o valor máximo). Explique como esse parâmetro influencia a geração dos marcadores, a separação de objetos adjacentes e a ocorrência de sobre-segmentação.

4. **(15%)** Utilizando `mm::drawImg`, construa uma demonstração visual passo a passo da erosão de uma imagem binária 7×7 com elemento estruturante quadrado 3×3. Para cada posição analisada, indique se o elemento estruturante está completamente contido no objeto e justifique o valor atribuído ao pixel de saída.

5. **(15%)** Demonstre experimentalmente a dualidade entre erosão e dilatação verificando a identidade $(A \ominus B)^c = A^c \oplus \hat{B}$ utilizando `mm::ero`, `mm::dil` e `mm::bnot`. Calcule a diferença pixel a pixel entre os dois lados da equação e apresente o resultado utilizando `mm::histImg` ou visualização equivalente.

6. **(15%)** Implemente manualmente o gradiente morfológico utilizando apenas `mm::ero` e `mm::dil`, comparando o resultado com `mm::gradm(img, B)`. Avalie o efeito de diferentes elementos estruturantes (quadrado 3×3, disco 5×5 e linha 1×9) sobre a detecção de bordas.

7. **(15%)** Construa um *pipeline* completo para contagem e classificação de moedas por tamanho (pequena, média e grande) utilizando área e circularidade como descritores. Gere uma máscara de referência (*ground truth*) manualmente e calcule a métrica IoU (*Intersection over Union*) para avaliar a qualidade da segmentação. Apresente os resultados em tabela e por meio de visualizações produzidas com `mm::show`.

Referências do Capítulo

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

- Gonzalez (2018) para os conceitos de segmentação, limiarização de Otsu, Transformada de Distância, *watershed*, morfologia matemática e descritores de forma.
- Matheron (1975) e Serra (1982) para a fundamentação teórica original, formulação algébrica e desenvolvimento da Morfologia Matemática.
- Szeliski (2022) para segmentação baseada em regiões, rotulação de componentes conexos, *watershed* baseado em marcadores e avaliação de segmentação por meio da métrica IoU.
- Bradski (2008) para a utilização prática da biblioteca OpenCV, incluindo funções como `mm::dist`, `mm::watershed`, `mm::label0` e extração de contornos.
- Redmon (2016) para a introdução aos detectores modernos da família YOLO e sua relação com descritores geométricos como *bounding boxes* extraídas por segmentação.
- Singh (2024) para a construção de labirintos complexos baseados em ciclos hamiltonianos sobre mosaicos quasicristalinos, utilizados como exemplo de aplicação da Transformada de Distância Geodésica e de algoritmos de busca de caminhos.
- Zampirolli (2025) para a implementação dos operadores morfológicos, transformadas geodésicas e resolução de labirintos por propagação de distâncias em domínios restritos.

4.9 💻 Parte Prática com Exercícios de Programação

Esta lista transforma os conceitos do Capítulo 4 em uma trilha prática de segmentação e morfologia matemática. Os EPs começam com limiarização e avançam até rotulação e descritores de componentes, sempre com matrizes pequenas para que cada pixel possa ser conferido à mão.

> ❗ Regra comum dos EPs morfológicos
>
> Nas operações com vizinhança, **não faça padding**. Para cada pixel, avalie apenas as posições do elemento estruturante que caem dentro do domínio da imagem. Essa é a mesma ideia das implementações didáticas em `morph.py`, como `mm::dil0`, `mm::ero0`, `mm::dil1` e `mm::label0`: a vizinhança é recortada pelo domínio válido da imagem.

🎯 Objetivo deste Caderno

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

Download

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

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # artefatos de build da trilha C++ (.cpp, binário,
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")

# O kernel é Python mesmo na trilha C++: `mm` (morph.py) é usado pelos
# simuladores, pela exibição das figuras que o binário C++ gera e pelo
# estado mm::Image entre células. cpp=True baixa também a trilha compilada
# (morph.hpp + stb_image*.h), usada no #include das células %%writefile *.cpp.
```

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

```
Ambiente pronto. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Executando os Testes

Para avaliar os testes, execute `TestSuite("EP04_01.extensão").run()` numa nova célula, trocando a extensão pela da linguagem usada (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). O sistema baixa os casos de teste do GitHub, executa o programa e calcula a nota automaticamente.

Para testar código Python diretamente, sem salvar arquivo, use `run_code(codigo)` passando o código como *string* numa variável `codigo`:

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

4.9.1 EP04_01 Limiarização Global por Limiar Fixo

Em **scanners de documentos** e em **sistemas de leitura de código de barras**, a primeira etapa do processamento é sempre separar o que é "objeto" (tinta, texto, barras) do que é "fundo" (papel, embalagem). A **limiarização global** faz exatamente isso: compara cada pixel a um único limiar T e decide, em tempo real, se ele pertence à classe clara ou à classe escura. É o operador de segmentação mais simples — e mesmo assim, está por trás de boa parte dos *pipelines* industriais de inspeção visual. Ver na Figura 4.30 uma simulação deste EP.

4.9.1.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros L (linhas) e C (colunas).
2. **Limiar:** Ler o inteiro T (limiar de decisão).
3. **Dados:** Ler os valores inteiros da matriz original linha a linha.
4. **Mapeamento:** Para cada pixel p, calcular o novo valor pela equação:

$$p' = \begin{cases} 255, & \text{se } p > T \\ 0, & \text{se } p \leq T \end{cases}$$

5. **Saída:** Exibir a matriz binarizada com dimensões $L \times C$.

4.9.1.2 📌 Restrições Computacionais

- **Binarização:** A saída contém **apenas** os valores 0 ou 255.
- **Comparação estrita:** O critério usa $> T$ (pixels iguais a T tornam-se fundo).
- **Tipo:** O resultado final deve ser inteiro.
- **Observação:** Este EP segue a convenção da OpenCV (`cv2.THRESL_BINARY`): apenas pixels com valor **maior que** T tornam-se brancos (`255`); pixels com valor **igual a** T permanecem pretos (`0`).

4.9.1.3 🧠 Fundamentação Teórica

Parâmetro	Tipo	Impacto Visual
T **pequeno**	Inteiro	A maioria dos pixels torna-se branca

Parâmetro	Tipo	Impacto Visual
T **grande**	Inteiro	A maioria dos pixels torna-se preta
T **bem escolhido**	Inteiro	Separa nitidamente objeto e fundo

4.9.1.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linha 3: Inteiro T.
- Linhas seguintes: Elementos inteiros da matriz original.

Saída:

- Matriz binarizada em L linhas e C colunas, valores 0 ou 255 separados por espaço.

4.9.1.5 📌 Exemplos

Entrada	Saída	Observação
2 4 100 0 99 100 180 255 30 120 80	0 0 0 255 255 0 255 0	$T = 100$: apenas pixels com valor maior que 100 tornam-se brancos; por isso, 99 e 100 tornam-se pretos.
1 3 0 0 50 255	0 255 255	$T = 0$: apenas pixels com valor estritamente maior que 0 tornam-se brancos.

```
%%writefile EP04_01.cpp
// sua solução
```

```
Overwriting EP04_01.cpp
```

```
TestSuite("EP04_01.cpp").run()
```

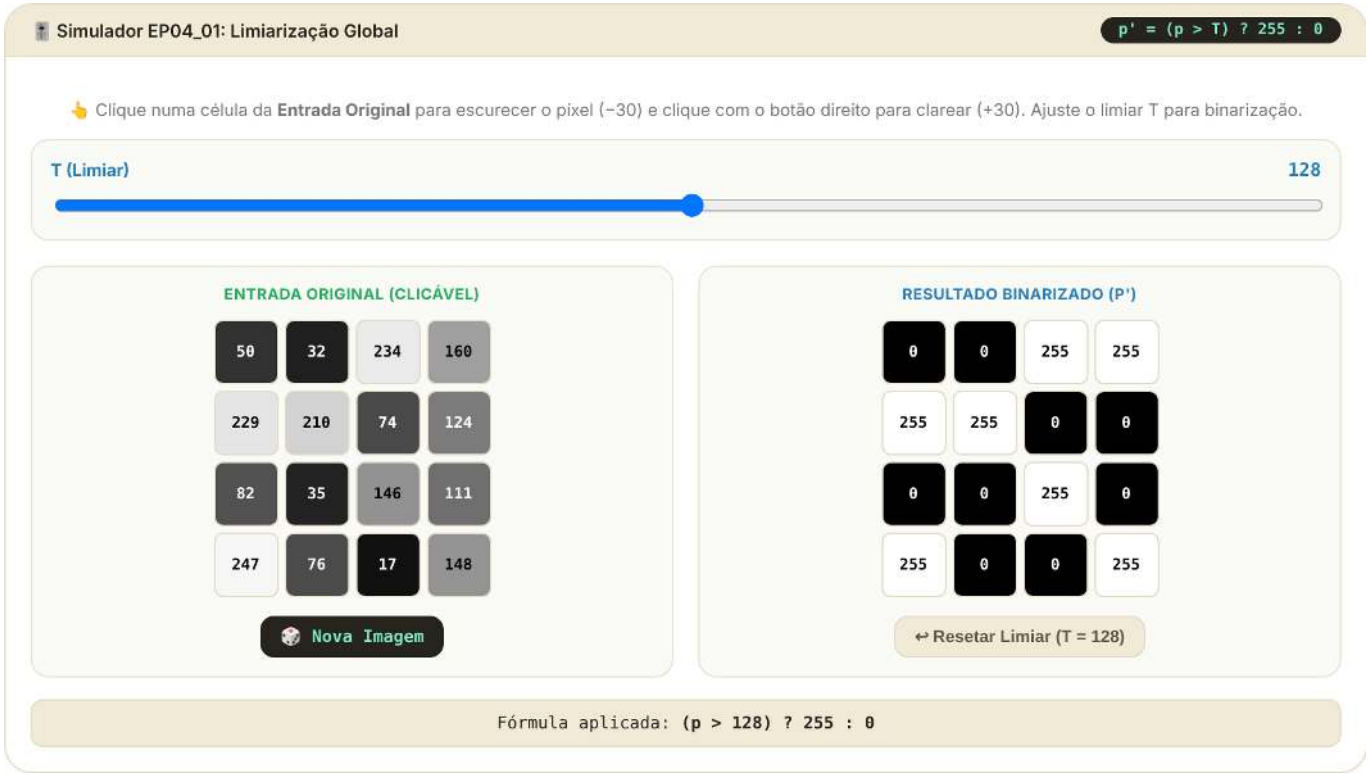
```
<IPython.core.display.HTML object>
```

4.9.2 EP04_02 📊 Limiarização Automática de Otsu

Escolher manualmente o limiar T funciona quando a iluminação é estável, mas em **microscopia digital** e em **inspeção de lâminas de sangue**, cada amostra tem um contraste diferente — um limiar fixo falharia de imagem para imagem. O **método de Otsu** resolve isso encontrando, sozinho, o limiar que **maximiza a separação estatística** entre as duas classes de pixels, tornando a segmentação automática e adaptativa. Ver na Figura 4.31 uma simulação deste EP.

4.9.2.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros L (linhas) e C (colunas).
2. **Dados:** Ler os valores inteiros da matriz original linha a linha.
3. **Histograma:** Construir o histograma $h[i]$, $i = 0, \ldots, 255$, contando quantos pixels têm valor i.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-ep0401-limiar.html

Figura 4.30: Simulador EP04_01: Limiarização Global por Limiar Fixo (p' = (p > T) ? 255 : 0)

4. **Busca do limiar:** Para cada candidato T de 1 a 255, calcular a **variância entre classes**:

$$\sigma_B^2(T) = \frac{n_0 \cdot n_1}{N^2}(m_0 - m_1)^2$$

onde n_0, n_1 são as quantidades de pixels com valor $< T$ e $\geq T$, m_0, m_1 são suas médias, e $N = L \times C$.

5. **Escolha:** O limiar ótimo T^* é o que maximiza $\sigma_B^2(T)$ (em caso de empate, manter o **primeiro** encontrado).

6. Aplicação: Binarizar a imagem usando T*, aplicando:

$$p' = \begin{cases} 255, & \text{se } p > T^* \\ 0, & \text{se } p \leq T^* \end{cases}$$

4.9.2.2 📌 Restrições Computacionais

- **Candidatos válidos:** Ignorar T que deixe $n_0 = 0$ ou $n_1 = 0$ (classe vazia).
- **Empate:** Sempre manter o **primeiro** T que atingiu o valor máximo de σ_B^2.
- **Tipo:** T^* e a matriz de saída devem ser inteiros.
- **Convenção OpenCV:** A binarização segue `cv2.THRESL_BINARY`; pixels com valor exatamente igual a T^* tornam-se pretos.

4.9.2.3 🧠 Fundamentação Teórica

Conceito	Significado	Impacto
$\sigma_B^2(T)$ **alta**	Classes bem separadas em T	T é um bom candidato a limiar
Histograma bimodal	Dois “morros” distintos	Otsu encontra o vale entre eles

Conceito	Significado	Impacto
Histograma unimodal	Um único "morro"	Otsu ainda escolhe *algum* T, mas a segmentação é pouco confiável

4.9.2.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linhas seguintes: Elementos inteiros da matriz original.

Saída:

- Matriz binarizada em L linhas e C colunas, valores 0 ou 255.

4.9.2.5 📌 Exemplos

Entrada	Saída	Observação
4 4 12 12 12 200 12 12 200 200 12 200 200 200 200 200 200 200	0 0 0 255 0 0 255 255 0 255 255 255 255 255 255 255	Histograma bimodal claro: 12 e 200
1 2 10 250	0 250	Apenas dois valores: T^* fica no maior

```
%%writefile EP04_02.cpp
// sua solução
```

```
Overwriting EP04_02.cpp
```

```
TestSuite("EP04_02.cpp").run()
```

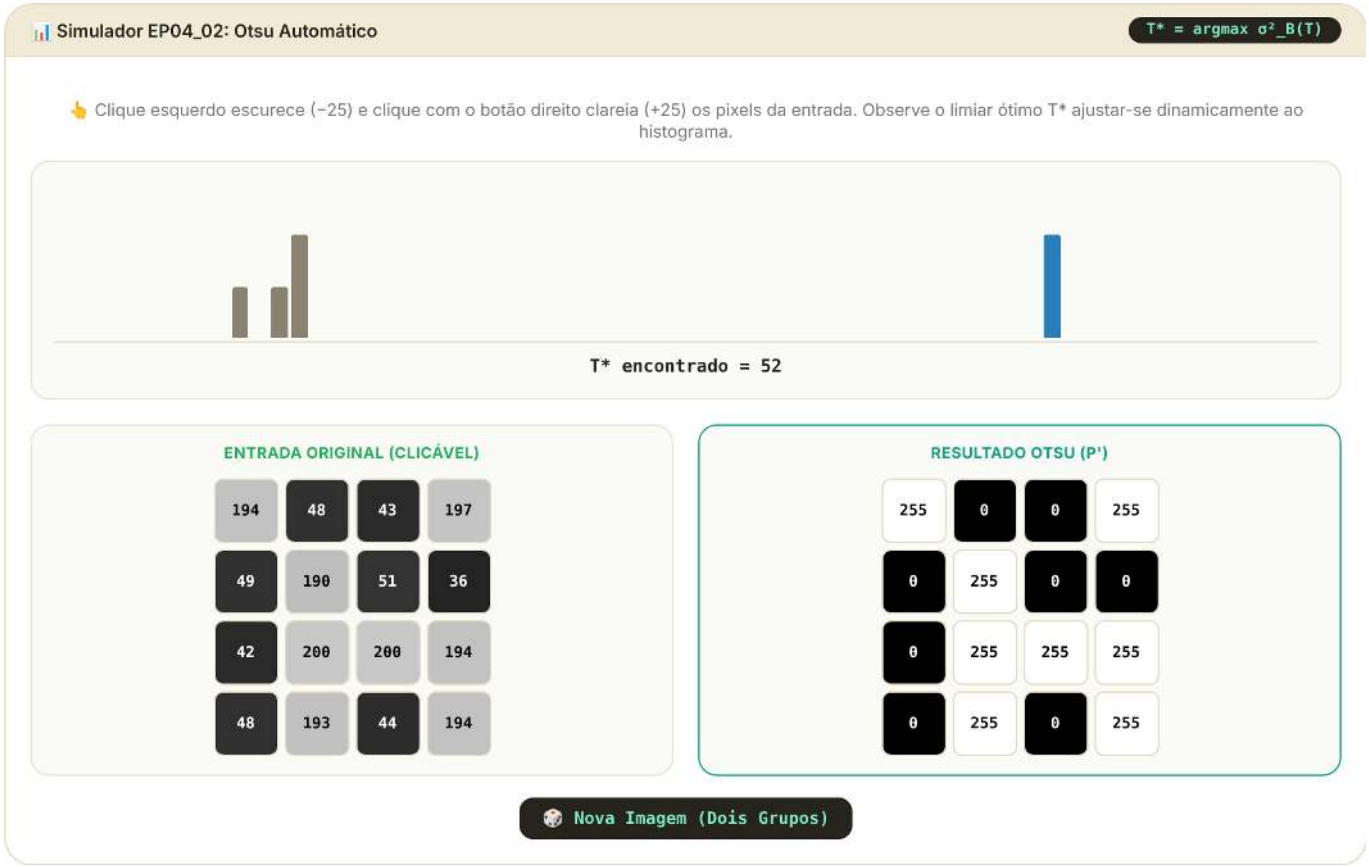
```
<IPython.core.display.HTML object>
```

4.9.3 EP04_03 🌱 Dilatação Binária Plana (mm.dil0)

Em **microscopia de partículas** e em **OCR de placas de carro desgastadas**, traços finos ou descontínuos precisam ser "engrossados" para que o reconhecimento funcione. A **dilatação morfológica** faz exatamente isso: expande regiões claras usando um elemento estruturante B — a mesma operação implementada em `morph.py` como `mm::dil0(f, B)`, usada quando B é **plano** (sem pesos, só 0/1). Ver na Figura 4.32 uma simulação deste EP.

4.9.3.1 📋 Diretrizes de Implementação

1. **Dimensões da imagem:** Ler os inteiros L (linhas) e C (colunas) de f.
2. **Dimensões de B:** Ler os inteiros L_B (linhas) e C_B (colunas) do elemento estruturante.
3. **Elemento estruturante:** Ler a matriz B com valores 0 ou 1, linha a linha.
4. **Dados:** Ler a matriz f (a imagem original), linha a linha.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-ep0402-otsu.html

Figura 4.31: Simulador EP04_02: Limiarização Automática de Otsu (T* = argmax ²_B(T))

5. **Reflexão:** Construir B_{ref}, a versão de B refletida em 180° (linhas e colunas invertidas) — exatamente como faz `mm::dilo` internamente.
6. **Vizinhança sem padding:** Para cada pixel (y, x), percorrer as posições (by, bx) de B_{ref} centradas em (y, x), usando o deslocamento

$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\frac{L_B}{2} + 0{,}5, \quad o_x = -\frac{C_B}{2} + 0{,}5$$

   Descartar todo (v_y, v_x) fora de $[0, L) \times [0, C)$ — **não preencher com zeros**.
7. **Mapeamento:** Calcular cada pixel de saída como o **máximo** entre $f(y, x)$ e todos os $f(v_y, v_x)$ válidos cuja posição correspondente em B_{ref} vale 1:

$$g(y, x) = \max\left(f(y, x), \max_{\substack{(v_y, v_x) \text{ válido} \\ B_{ref}(by, bx)=1}} f(v_y, v_x)\right)$$

8. **Saída:** Exibir a matriz g com dimensões $L \times C$.

4.9.3.2 📌 Restrições Computacionais

- **Sem padding:** Jamais inventar vizinhos fora da imagem; usar apenas os que existem de fato.
- **Reflexão obrigatória:** B deve ser refletido antes de aplicado (é o que diferencia `mm::dilo` de uma simples busca por máximo).
- **Robustez de borda:** Se nenhuma posição válida de $B_{ref} = 1$ cair dentro do domínio para um dado pixel, ele **mantém seu valor original**.

4.9.3.3 🧠 Fundamentação Teórica

Conceito	Significado	Impacto Visual
Dilatação	$g \geq f$ sempre (extensiva)	Regiões claras crescem, buracos escuros encolhem
B **maior**	Vizinhança mais ampla	Crescimento mais agressivo
Reflexão de B	$B_{ref}(y,x) = B(-y,-x)$	Garante a definição formal de Minkowski da dilatação

4.9.3.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linha 3: Inteiro L_B.
- Linha 4: Inteiro C_B.
- Próximas L_B linhas: elementos inteiros (0 ou 1) da matriz B.
- Próximas L linhas: elementos inteiros da matriz f.

Saída:

- Matriz g em L linhas e C colunas, valores inteiros separados por espaço.

4.9.3.5 📌 Exemplos

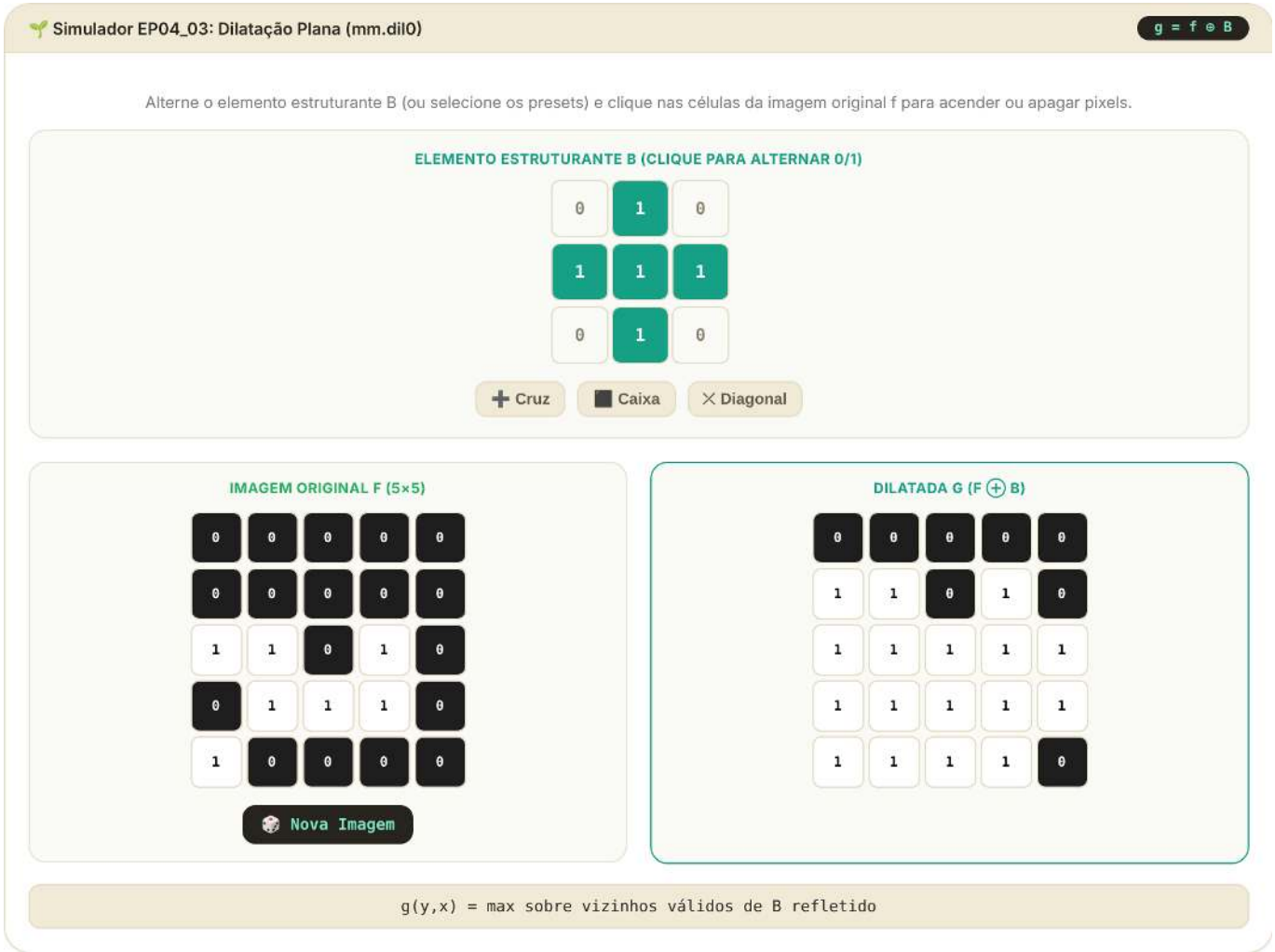
Entrada	Saída	Observação
3 3 3 3 0 1 0 1 1 1 0 1 0 0 0 0 0 9 0 0 0 0	0 9 0 9 9 9 0 9 0	B em cruz simétrico: ponto isolado se expande em cruz
1 4 1 3 1 1 1 10 200 5 80	200 200 200 80	B horizontal: cada pixel "puxa" o máximo dos vizinhos da linha

```
%%writefile EP04_03.cpp
// sua solução
```

```
Overwriting EP04_03.cpp
```

```
TestSuite("EP04_03.cpp").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-ep0403-dilatacao.html

Figura 4.32: Simulador EP04_03: Dilatação Binária Plana (g = f B)

4.9.4 EP04_04 🪨 Erosão Binária Plana (mm.ero0)

Se a dilatação engrossa, a **erosão** afina. Em **sistemas de contagem de células**, ela é usada para **separar células que se tocam**: ao "comer" as bordas de cada região, conexões finas entre objetos desaparecem antes mesmo de qualquer contagem ser feita. Em `morph.py`, essa é a operação `mm::ero0(f, B)` — a **dual** exata da dilatação, e a única das duas que **não** reflete o elemento estruturante. Ver na Figura 4.33 uma simulação deste EP.

4.9.4.1 📋 Diretrizes de Implementação

1. **Dimensões da imagem:** Ler os inteiros L (linhas) e C (colunas) de f.
2. **Dimensões de B:** Ler os inteiros L_B (linhas) e C_B (colunas) do elemento estruturante.
3. **Elemento estruturante:** Ler a matriz B com valores 0 ou 1, linha a linha.
4. **Dados:** Ler a matriz f (a imagem original), linha a linha.
5. **Vizinhança sem padding (sem reflexão!):** Para cada pixel (y, x), percorrer as posições (by, bx) de B **na ordem original** (sem refletir), usando o mesmo deslocamento do EP04_03:

$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\frac{L_B}{2} + 0{,}5, \quad o_x = -\frac{C_B}{2} + 0{,}5$$

Descartar todo (v_y, v_x) fora de $[0, L) \times [0, C)$. 6. **Mapeamento:** Calcular cada pixel de saída como o **mínimo** entre $f(y, x)$ e todos os $f(v_y, v_x)$ válidos cuja posição correspondente em B vale 1:

$$g(y, x) = \min\left(f(y, x), \min_{\substack{(v_y, v_x)\ \text{válido} \\ B(by, bx)=1}} f(v_y, v_x)\right)$$

7. **Saída:** Exibir a matriz g com dimensões $L \times C$.

4.9.4.2 📌 Restrições Computacionais

- **Sem reflexão:** Diferente da dilatação, B é usado **exatamente como lido** — refletir aqui seria um erro conceitual grave.
- **Sem padding:** Vizinhos fora da imagem são simplesmente ignorados, nunca tratados como 0.
- **Robustez de borda:** Se nenhuma posição válida de $B = 1$ cair dentro do domínio, o pixel mantém seu valor original.

4.9.4.3 🧠 Fundamentação Teórica

Conceito	Significado	Impacto Visual
Erosão	$g \leq f$ sempre (anti-extensiva)	Regiões claras encolhem, ruído pontual desaparece
Dualidade	$\text{ero}(f, B) = -\text{dil}(-f, B_{ref})$	Erosão e dilatação são "espelhos" matemáticos
B **maior**	Erosão mais agressiva	Objetos finos somem completamente

4.9.4.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

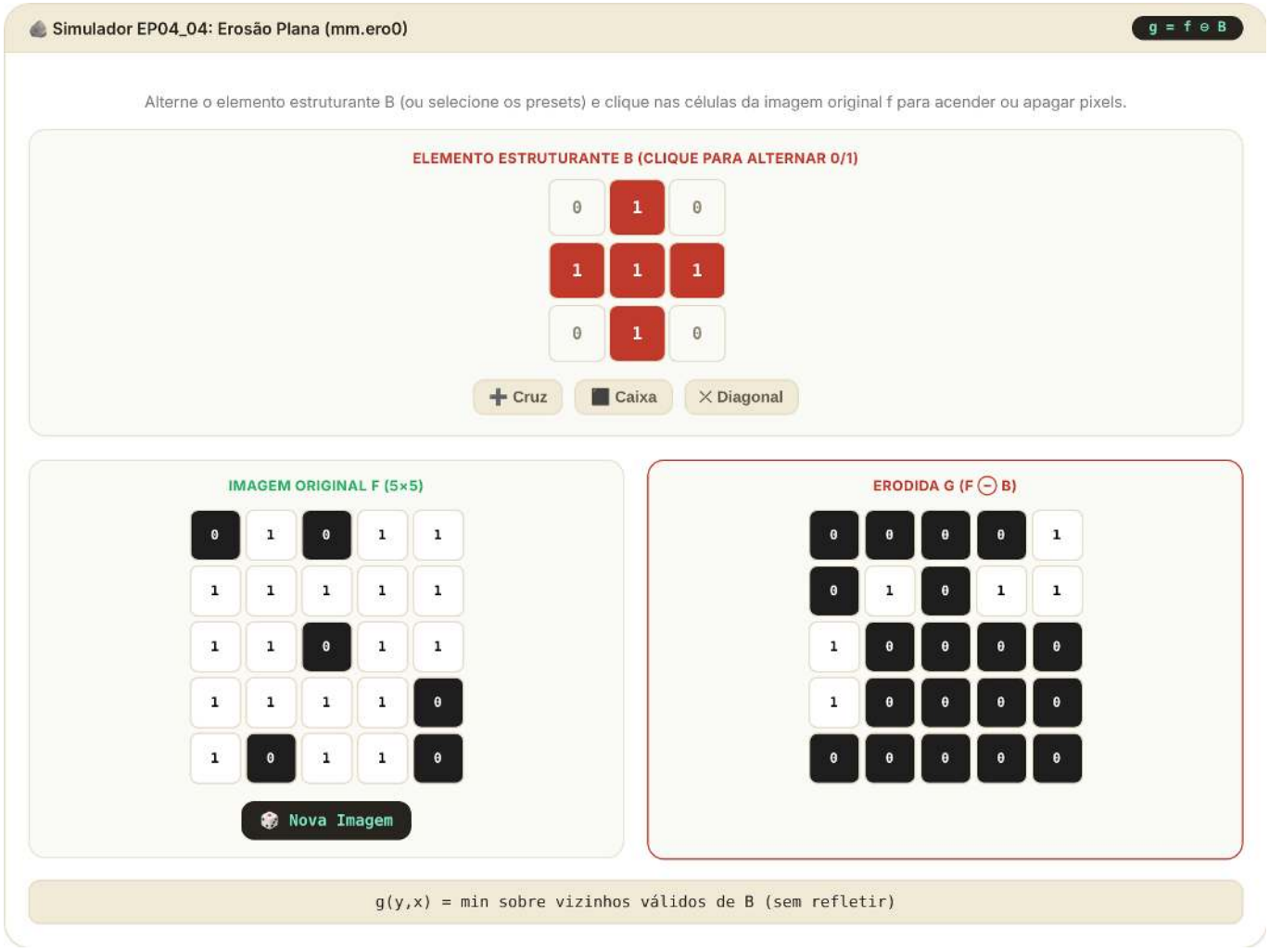
- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linha 3: Inteiro L_B.
- Linha 4: Inteiro C_B.
- Próximas L_B linhas: elementos inteiros (0 ou 1) da matriz B.
- Próximas L linhas: elementos inteiros da matriz f.

Saída:

- Matriz g em L linhas e C colunas, valores inteiros separados por espaço.

4.9.4.5 📌 Exemplos

Entrada	Saída	Observação
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	O “buraco” central (0) se propaga em cruz
1 4 1 3 1 1 1 10 200 5 80	10 5 5 80	B horizontal: cada pixel “puxa” o mínimo dos vizinhos da linha



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-ep0404-erosao.html

Figura 4.33: Simulador EP04_04: Erosão Binária Plana (g = f B)

```
%%writefile EP04_04.cpp
// sua solução
```

```
Overwriting EP04_04.cpp
```

```
TestSuite("EP04_04.cpp").run()
```

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

4.9.5 EP04_05 🧹 Abertura Morfológica (Remoção de Ruído)

Imagens capturadas por **sensores de baixo custo**, como os de drones agrícolas, costumam vir salpicadas de pequenos pontos de ruído — pixels isolados que não representam nada real. Aplicar erosão seguida de dilatação com o **mesmo** elemento estruturante produz a **abertura**: ela "limpa" pontos e protuberâncias finas, mas devolve ao objeto principal praticamente seu tamanho original. É a combinação clássica usada em **pré-processamento de imagens de satélite** antes de qualquer contagem de área plantada. Ver na Figura 4.34 uma simulação deste EP.

4.9.5.1 📋 Diretrizes de Implementação

1. **Dimensões da imagem:** Ler os inteiros L (linhas) e C (colunas) de f.
2. **Dimensões de B:** Ler os inteiros L_B (linhas) e C_B (colunas) do elemento estruturante.
3. **Elemento estruturante:** Ler a matriz B com valores 0 ou 1, linha a linha.
4. **Dados:** Ler a matriz binária f (valores 0 ou 1), linha a linha.
5. **Erosão:** Calcular $e = f \ominus B$, usando exatamente o algoritmo do EP04_04 (sem refletir B, sem padding).
6. **Dilatação:** Calcular $g = e \oplus B$, usando exatamente o algoritmo do EP04_03 (refletindo B, sem padding) — mas agora aplicado sobre e, não sobre f.
7. **Saída:** Exibir a matriz resultante g (a **abertura** de f por B) com dimensões $L \times C$.

4.9.5.2 📌 Restrições Computacionais

- **Ordem fixa:** É **sempre** erosão primeiro, depois dilatação — a ordem inversa define outro operador (fechamento, do próximo EP).
- **Mesmo B:** O elemento estruturante usado na erosão e na dilatação deve ser idêntico.
- **Sem padding em nenhuma das duas etapas.**

4.9.5.3 🧠 Fundamentação Teórica

Conceito	Significado	Impacto Visual
Anti-extensividade	$g \subseteq f$ sempre	A abertura nunca cria pixel novo, só remove
Idempotência	abertura(abertura(f)) = abertura(f)	Aplicar de novo não muda mais nada
Pontos isolados	Menores que B	São completamente eliminados
Núcleo do objeto	Maior que B	É recuperado quase intacto pela dilatação final

4.9.5.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linha 3: Inteiro L_B.

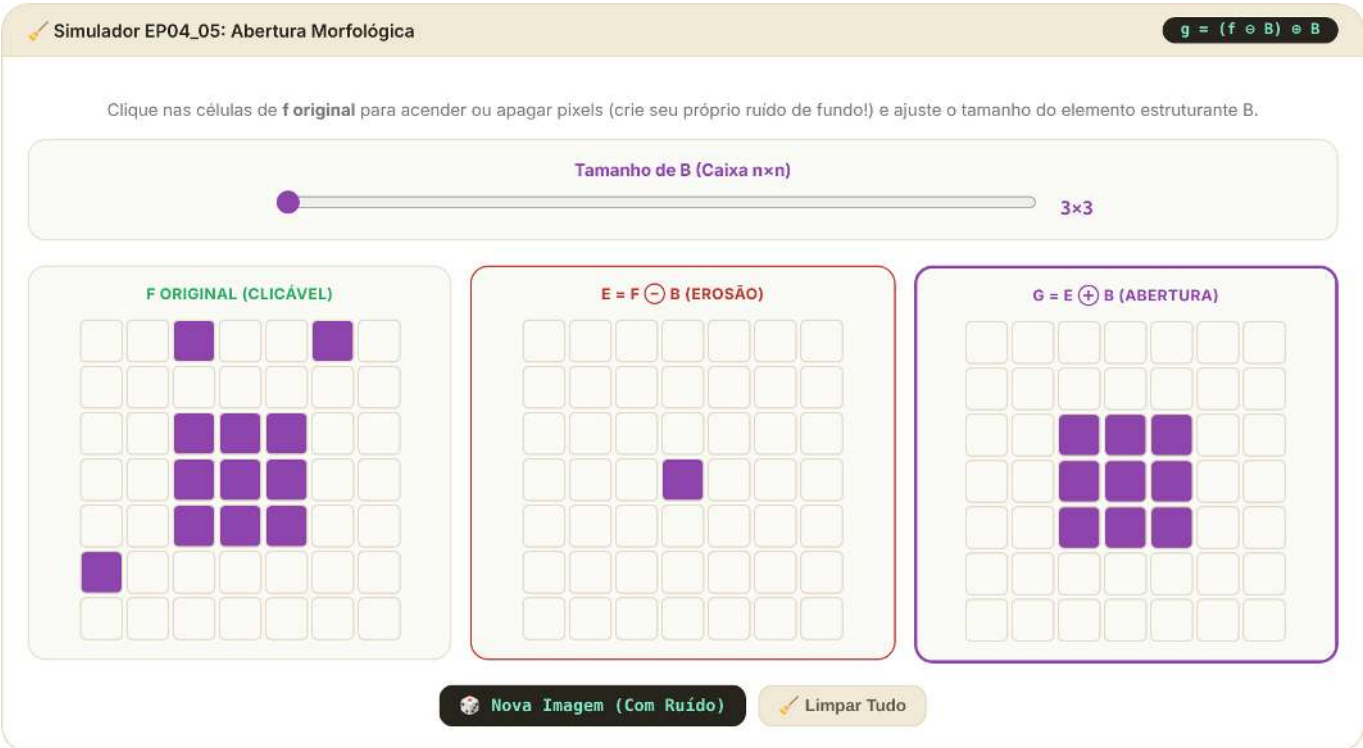
- Linha 4: Inteiro C_B.
- Próximas L_B linhas: elementos inteiros (0 ou 1) da matriz B.
- Próximas L linhas: elementos inteiros (0 ou 1) da matriz f.

Saída:

- Matriz resultante em L linhas e C colunas, valores 0 ou 1.

4.9.5.5 📌 Exemplos

Entrada	Saída	Observação
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	Pontos isolados e a protuberância fina desaparecem; o quadrado central sobrevive



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-ep0405-abertura.html

Figura 4.34: Simulador EP04_05: Abertura Morfológica (g = (f B) B)

```
%%writefile EP04_05.cpp
// sua solução
```

```
Overwriting EP04_05.cpp
```

```
TestSuite("EP04_05.cpp").run()
```

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

4.9.6 EP04_06 🧩 Fechamento Morfológico (Preenchimento de Falhas)

Em **digitalização de impressões digitais**, sulcos da pele às vezes ficam interrompidos por sujeira ou ressecamento, criando pequenas falhas na curva contínua que deveria existir. O **fechamento** — dilatação seguida de erosão com o mesmo elemento estruturante — é o operador dual da abertura: ele **preenche buracos pequenos e reentrâncias estreitas**, sem alterar significativamente o contorno externo do objeto. É a etapa padrão antes de extrair o esqueleto de uma impressão digital. Ver na Figura 4.35 uma simulação deste EP.

4.9.6.1 📋 Diretrizes de Implementação

1. **Dimensões da imagem:** Ler os inteiros L (linhas) e C (colunas) de f.
2. **Dimensões de B:** Ler os inteiros L_B (linhas) e C_B (colunas) do elemento estruturante.
3. **Elemento estruturante:** Ler a matriz B com valores 0 ou 1, linha a linha.
4. **Dados:** Ler a matriz binária f (valores 0 ou 1), linha a linha.
5. **Dilatação:** Calcular $d = f \oplus B$, usando exatamente o algoritmo do EP04_03 (refletindo B, sem padding).
6. **Erosão:** Calcular $g = d \ominus B$, usando exatamente o algoritmo do EP04_04 (sem refletir B, sem padding) — agora aplicado sobre d, não sobre f.
7. **Saída:** Exibir a matriz resultante g (o **fechamento** de f por B) com dimensões $L \times C$.

4.9.6.2 📌 Restrições Computacionais

- **Ordem fixa:** É **sempre** dilatação primeiro, depois erosão — a ordem inversa é a abertura do EP04_05.
- **Mesmo B:** O elemento estruturante usado na dilatação e na erosão deve ser idêntico.
- **Sem padding em nenhuma das duas etapas.**

4.9.6.3 🧠 Fundamentação Teórica

Conceito	Significado	Impacto Visual
Extensividade	$g \supseteq f$ sempre	O fechamento nunca remove pixel, só adiciona
Idempotência	$\text{fecha}(\text{fecha}(f)) = \text{fecha}(f)$	Aplicar de novo não muda mais nada
Buracos pequenos	Menores que B	São completamente preenchidos
Dualidade	$\text{fecha}(f) = \overline{\text{abre}(\bar{f})}$	É a abertura aplicada ao "negativo" da imagem

4.9.6.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

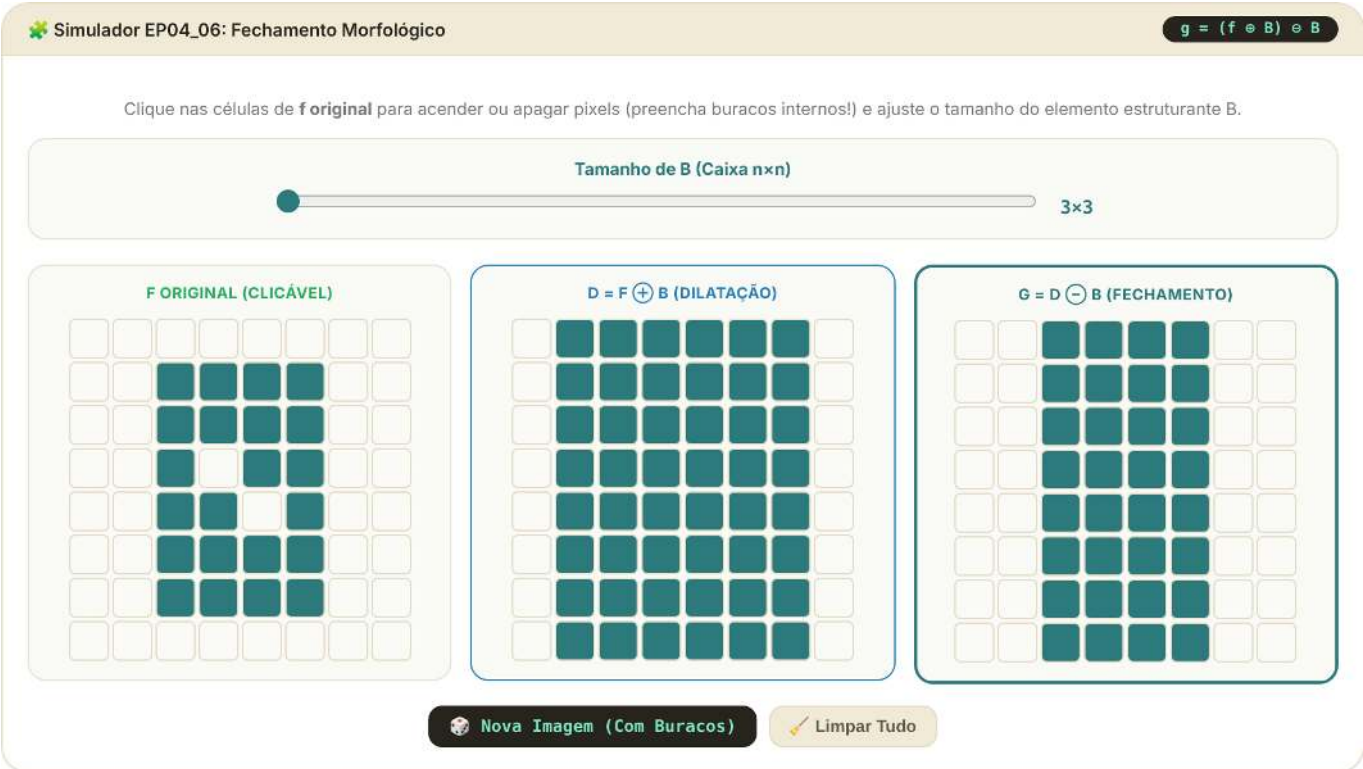
- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linha 3: Inteiro L_B.
- Linha 4: Inteiro C_B.
- Próximas L_B linhas: elementos inteiros (0 ou 1) da matriz B.
- Próximas L linhas: elementos inteiros (0 ou 1) da matriz f.

Saída:

- Matriz resultante em L linhas e C colunas, valores 0 ou 1.

4.9.6.5 📌 Exemplos

Entrada	Saída	Observação
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	Os dois buracos internos não-adjacentes são totalmente preenchidos



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-ep0406-fechamento.html

Figura 4.35: Simulador EP04_06: Fechamento Morfológico (g = (f B) B)

```
%%writefile EP04_06.cpp
// sua solução
```

```
Overwriting EP04_06.cpp
```

```
TestSuite("EP04_06.cpp").run()
```

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

4.9.7 EP04_07 Dilatação e Erosão com Pesos (mm.dil1 / mm.ero1)

Até agora, o elemento estruturante só dizia "este vizinho conta" ou "não conta" — mas em **modelos digitais de elevação** (usados em SIG e em planejamento de drenagem urbana), cada vizinho deveria ter um **peso diferente** dependendo da distância ou da direção do relevo. As versões **ponderadas** da dilatação e da erosão, implementadas em `morph.py` como `mm::dil1(f, b)` e `mm::ero1(f, b)`, somam (ou subtraem) o peso de cada vizinho antes de tomar o máximo (ou mínimo) — generalizando tudo o que foi feito nos EPs anteriores. Ver na Figura 4.36 uma simulação deste EP.

4.9.7.1 📋 Diretrizes de Implementação

1. **Dimensões da imagem:** Ler os inteiros L (linhas) e C (colunas) de f.
2. **Dimensões de b:** Ler os inteiros L_B (linhas) e C_B (colunas) do elemento estruturante ponderado.
3. **Pesos:** Ler a matriz b de pesos **inteiros** (podem ser negativos, zero ou positivos), linha a linha.
4. **Dados:** Ler a matriz f (a imagem original), linha a linha.
5. **Vizinhança sem padding:** Para cada pixel (y, x), percorrer **todas** as posições (by, bx) de b (não apenas onde valeria 1 — aqui **todo** peso participa), usando o mesmo deslocamento dos EPs anteriores:

$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\tfrac{L_B}{2} + 0{,}5, \quad o_x = -\tfrac{C_B}{2} + 0{,}5$$

   Descartar todo (v_y, v_x) fora de $[0, L) \times [0, C)$.
6. **Dilatação ponderada:** Calcular

$$g_{dil}(y, x) = \max\left(f(y, x),\ \max_{(v_y, v_x)\ \text{válido}} \left(f(v_y, v_x) + b(by, bx)\right)\right)$$

7. **Erosão ponderada:** Calcular, **usando o mesmo b e sem refletir**:

$$g_{ero}(y, x) = \min\left(f(y, x),\ \min_{(v_y, v_x)\ \text{válido}} \left(f(v_y, v_x) - b(by, bx)\right)\right)$$

8. **Saída:** Exibir **primeiro** a matriz g_{dil} completa, e **depois** a matriz g_{ero} completa.

4.9.7.2 📌 Restrições Computacionais

- **Nenhuma das duas reflete** b — a versão ponderada não usa reflexão, mesmo na dilatação (diferente de `mm::dil0`).
- **Todos os pesos participam:** Não existe aqui o filtro "$B = 1$"; mesmo peso 0 entra na conta.
- **Sem padding:** vizinhos fora da imagem são ignorados, nunca virtualmente preenchidos.
- **Tipo:** A saída pode conter valores negativos ou maiores que 255 — **não** há *clipping* neste EP.
- **Dica:** Para remover mensagens de overflow ao ultrapassar limites do tipo uint8, incluir no início do código:

```
import warnings
warnings.filterwarnings("ignore")
```

4.9.7.3 🧠 Fundamentação Teórica

Conceito	Significado	Impacto Visual
Peso positivo	"Puxa" o valor do vizinho para cima na dilatação	Simula relevo que sobe naquela direção

Conceito	Significado	Impacto Visual
Peso negativo	Reduz a contribuição do vizinho	Simula distância ou atenuação direcional
Dualidade ponderada	$\text{ero1}(f,b) = -\text{dil1}(-f,b)$	A simetria entre as duas operações se mantém mesmo com pesos

4.9.7.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linha 3: Inteiro L_B.
- Linha 4: Inteiro C_B.
- Próximas L_B linhas: elementos inteiros (podem ser negativos) da matriz b.
- Próximas L linhas: elementos inteiros da matriz f.

Saída:

- Primeiro a matriz g_{dil} em L linhas e C colunas.
- Em seguida a matriz g_{ero} em L linhas e C colunas.

4.9.7.5 📌 Exemplos

Entrada	Saída	Observação
3 3 3 3 0 1 0 1 2 1 0 1 0 10 20 30 40 50 60 70 80 90	50 60 61 80 90 91 81 91 92 8 9 19 9 10 20 39 40 50	Peso central 2 acelera o crescimento na dilatação e o encolhimento na erosão

```
%%writefile EP04_07.cpp
// sua solução
```

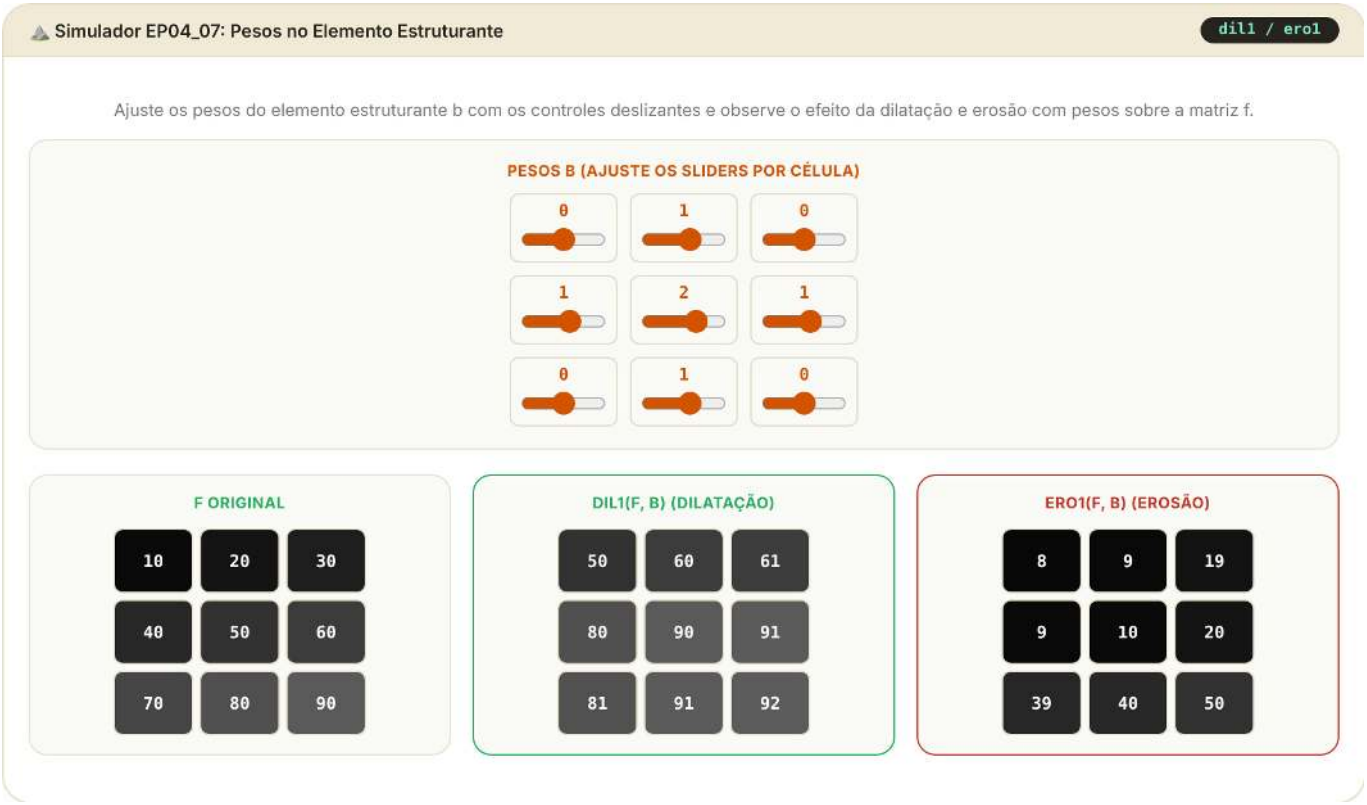
```
Overwriting EP04_07.cpp
```

```
TestSuite("EP04_07.cpp").run()
```

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

4.9.8 EP04_08 🌋 Gradiente Morfológico, Top-hat e Black-hat

Em **inspeção automática de placas de circuito**, três perguntas aparecem o tempo todo: onde estão as **bordas** dos componentes? Quais **detalhes claros e pequenos** (como pontos de solda) se destacam do fundo? Quais **reentrâncias escuras** (como fissuras) o fundo esconde? Um único par erosão/dilatação responde às três: o **gradiente morfológico** evidencia contornos, o **top-hat** revela picos estreitos, e o **black-hat** revela vales estreitos — três ferramentas, uma só vizinhança. Ver na Figura 4.37 uma simulação deste EP.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-ep0407-pesos.html

Figura 4.36: Simulador EP04_07: Dilatação e Erosão com Pesos (mm.dil1 / mm.ero1)

4.9.8.1 📋 Diretrizes de Implementação

1. **Dimensões da imagem:** Ler os inteiros L (linhas) e C (colunas) de f.
2. **Dimensões de B:** Ler os inteiros L_B (linhas) e C_B (colunas) do elemento estruturante.
3. **Elemento estruturante:** Ler a matriz B com valores 0 ou 1, linha a linha.
4. **Dados:** Ler a matriz f (a imagem original, em tons de cinza), linha a linha.
5. **Operadores de base:** Calcular, exatamente como nos EPs 04_03 a 04_06:
   - $d = f \oplus B$ (dilatação),
   - $e = f \ominus B$ (erosão),
   - abertura $= e \oplus B$,
   - fechamento $= d \ominus B$.
6. **Gradiente morfológico:** $\text{grad}(y,x) = d(y,x) - e(y,x)$.
7. **Top-hat:** $\text{tophat}(y,x) = f(y,x) - \text{abertura}(y,x)$.
8. **Black-hat:** $\text{blackhat}(y,x) = \text{fechamento}(y,x) - f(y,x)$.
9. **Saída:** Exibir, **nesta ordem**, as três matrizes completas: gradiente, top-hat, black-hat.

4.9.8.2 📌 Restrições Computacionais

- **Sem padding em nenhuma etapa intermediária** — dilatação, erosão, abertura e fechamento seguem as mesmas regras de vizinhança dos EPs anteriores.
- **Não há *clipping*:** as três saídas podem conter qualquer valor inteiro (o gradiente é sempre ≥ 0, mas top-hat e black-hat também).
- **Reaproveitamento:** d e e devem ser calculados **uma única vez** e reaproveitados para montar abertura, fechamento e gradiente.

4.9.8.3 🧠 Fundamentação Teórica

Operador	Fórmula	O que revela
Gradiente	$d - e$	Bordas: zero em regiões planas, alto nas transições
Top-hat	$f - \text{abertura}(f)$	Elementos **claros e finos**, menores que B
Black-hat	$\text{fechamento}(f) - f$	Elementos **escuros e finos**, menores que B

4.9.8.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linha 3: Inteiro L_B.
- Linha 4: Inteiro C_B.
- Próximas L_B linhas: elementos inteiros (0 ou 1) da matriz B.
- Próximas L linhas: elementos inteiros da matriz f.

Saída:

- Matriz gradiente em L linhas e C colunas.
- Matriz top-hat em L linhas e C colunas.
- Matriz black-hat em L linhas e C colunas.

4.9.8.5 📌 Exemplos

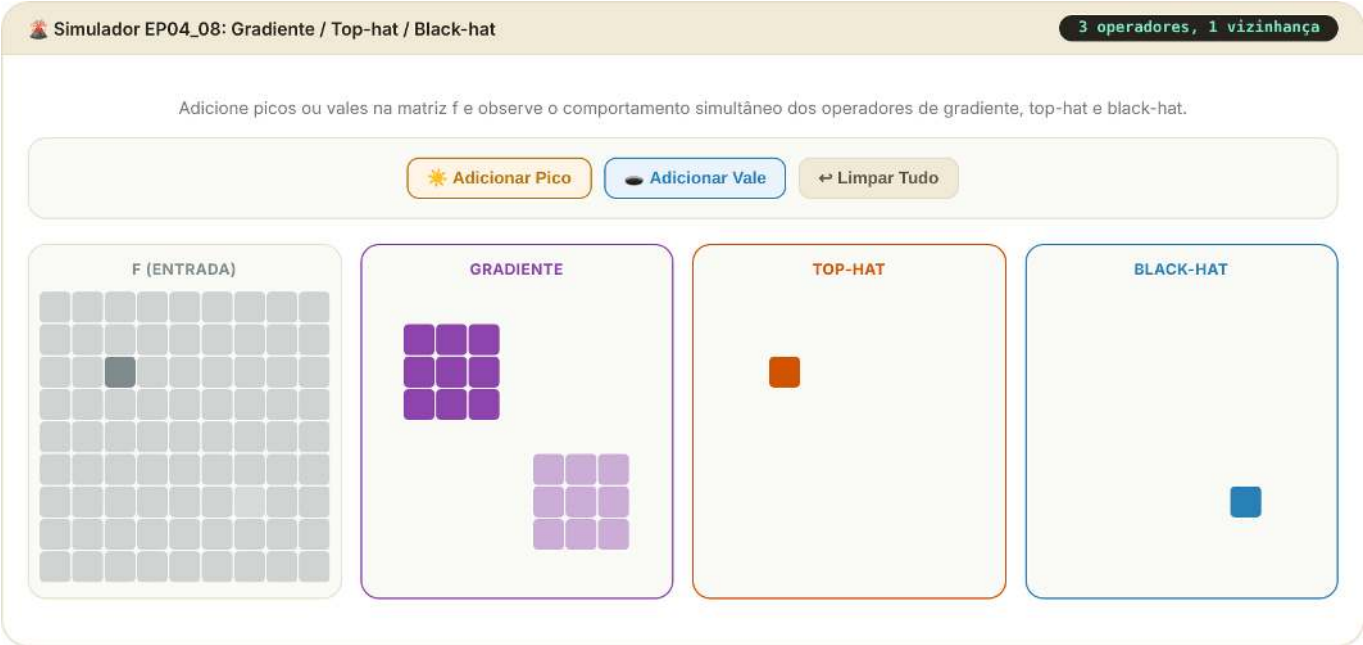
Entrada	Saída	Observação
9 9 3 3 1 1 1 1 1 1 1 1 1 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 80 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 2 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10	(gradiente: halo $3 \times 3 = 70$ em torno de $(2, 2)$ e halo $3 \times 3 = 8$ em torno de $(6, 6)$, resto 0) (top-hat: único 70 em $(2, 2)$, resto 0) (black-hat: único 8 em $(6, 6)$, resto 0)	Pico isolado vira top-hat; vale isolado vira black-hat; ambos aparecem no gradiente

```
%%writefile EP04_08.cpp
// sua solução
```

```
Overwriting EP04_08.cpp
```

```
TestSuite("EP04_08.cpp").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-ep0408-gradiente.html

Figura 4.37: Simulador EP04_08: Gradiente Morfológico, Top-hat e Black-hat

4.9.9 EP04_09 Transformada de Distância e o "Miolo" do Objeto

Em **robótica móvel**, ao planejar uma rota dentro de um corredor, o robô quer saber não apenas *onde* existe espaço livre, mas também **quão longe** cada ponto livre está da parede mais próxima. Os caminhos mais seguros tendem a passar pelo "miolo" do corredor, longe dos obstáculos.

A **transformada de distância morfológica** atribui a cada pixel um valor que representa sua distância até a borda mais próxima, segundo a métrica definida pelo elemento estruturante. Pixels próximos à borda recebem valores baixos, enquanto pixels mais internos recebem valores maiores. O pixel de valor máximo corresponde à região mais protegida do objeto, frequentemente associada ao seu centro morfológico.

Ver na Figura 4.38 uma simulação deste EP.

4.9.9.1 📋 Diretrizes de Implementação

1. **Dimensões da imagem:** ler os inteiros L (linhas) e C (colunas) da imagem f.
2. **Dimensões de B:** ler os inteiros L_B (linhas) e C_B (colunas) do elemento estruturante.
3. **Elemento estruturante:** ler a matriz b, contendo valor 0 no centro e valores negativos nas demais posições.
4. **Imagem:** ler a matriz binária f (valores 0 ou 1), linha a linha.
5. **Preparação:** multiplicar a imagem por $L \times C$, garantindo que os pixels internos tenham valor inicial suficientemente alto para a propagação das distâncias.
6. **Transformada de distância:** calcular a matriz de distâncias utilizando o método `mm::dist1(f,b)`.
7. **Saída:** exibir a matriz resultante da transformada de distância.

4.9.9.2 📌 Restrições Computacionais

- Utilizar a implementação de erosão ponderada fornecida pela biblioteca.
- O elemento estruturante pode conter valores negativos arbitrários.
- A transformada deve ser obtida pela aplicação iterativa de erosões ponderadas até atingir um ponto fixo.

⚠️ **Nota Crucial sobre Leitura de Matrizes:** Como o elemento estruturante pode conter valores inteiros negativos (por exemplo, `-1` e `-99`), **não utilize a função `mm::readImg` para ler a matriz** b. Essa função converte os dados para o tipo `uint8`, provocando *underflow* e corrompendo os valores negativos. Leia as L_B

linhas de b manualmente utilizando o tipo padrão `int`. A imagem f pode continuar sendo lida normalmente por `mm::readImg`.

4.9.9.3 🧠 Fundamentação Teórica

Conceito	Significado	Impacto Visual
dist(y, x)	Distância morfológica até a borda mais próxima segundo a métrica definida por b	Pixels mais internos recebem valores maiores
Valor máximo	Pixel mais distante da borda	Aproxima o centro morfológico do objeto
Elemento estruturante ponderado	Define os custos de deslocamento entre pixels vizinhos	Determina a métrica de distância utilizada
Objetos finos	Regiões estreitas do objeto	Produzem valores baixos de distância

4.9.9.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: inteiro L.
- Linha 2: inteiro C.
- Linha 3: inteiro L_B.
- Linha 4: inteiro C_B.
- Próximas L_B linhas: elementos inteiros da matriz b.
- Próximas L linhas: elementos binários (0 ou 1) da matriz f.

⚠️ **Nota de implementação:** Os elementos da matriz f (0 ou 1) devem ser multiplicados por **255** para gerar uma imagem binária adequada (0 e 255) antes de aplicar a Transformada de Distância (TD).

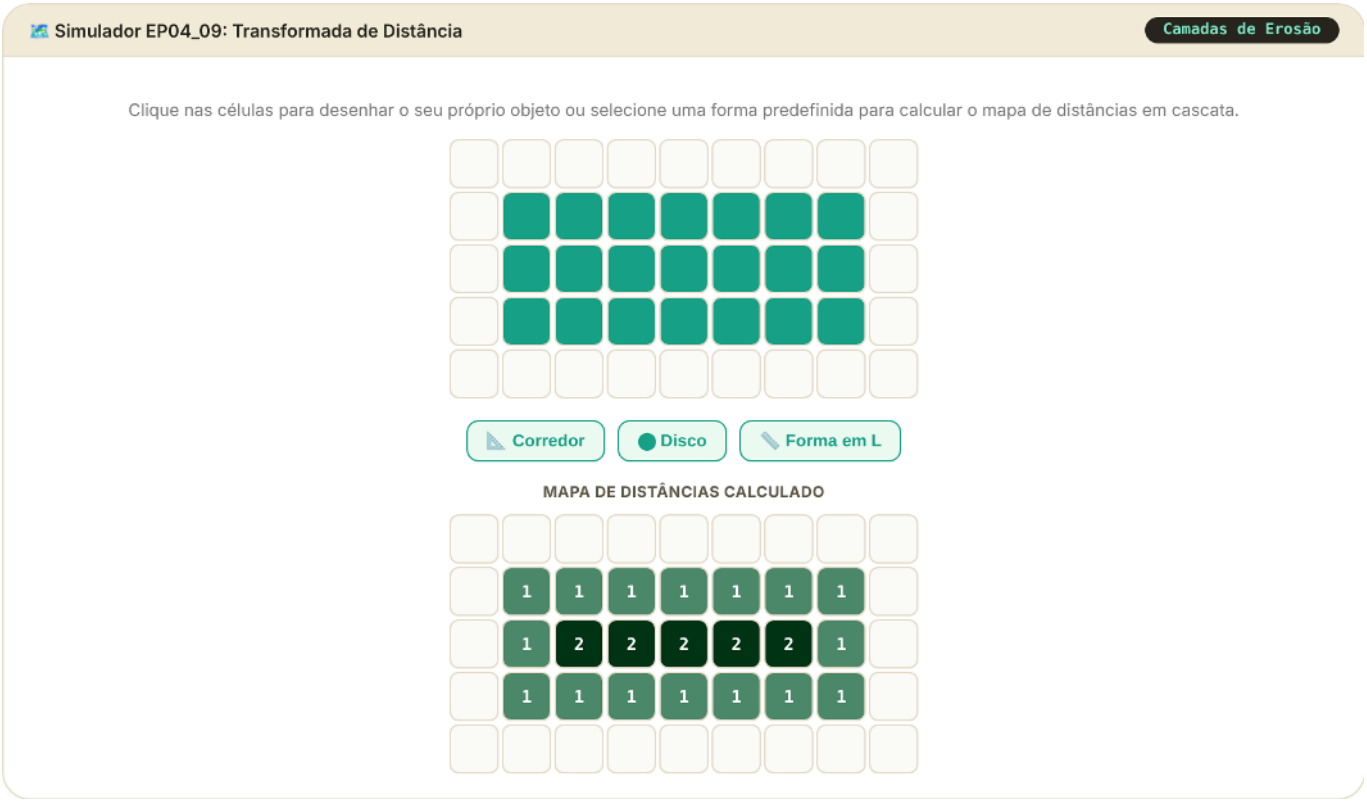
Saída:

- Matriz da transformada de distância em L linhas e C colunas.

4.9.9.5 📌 Exemplo

Entrada	Saída	Observação
5 9 3 3 -99 -1 -99 -1 0 -1 -99 -1 -99 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 2 2 2 2 2 1 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0	Resultado da transformada de distância.

Nota: o valor `-99` atua como uma aproximação prática de $-\infty$, impedindo a propagação pelas diagonais. Dessa forma, apenas os vizinhos horizontal e vertical contribuem para a distância, produzindo a distância de Manhattan.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-ep0409-distancia.html

Figura 4.38: Simulador EP04_09: Transformada de Distância (Camadas de Erosão)

```
%%writefile EP04_09.cpp
// sua solução
```

```
Overwriting EP04_09.cpp
```

```
TestSuite("EP04_09.cpp").run()
```

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

4.9.10 EP04_10 Separação de *Blobs*, Rotulação e Descritores

Em uma **linha de produção de moedas**, é comum que peças encostem umas nas outras na esteira, formando uma única mancha conectada na imagem — uma contagem ingênua erraria o total. A solução clássica combina operações morfológicas e análise de conectividade: primeiro uma **erosão** reduz ou rompe conexões frágeis entre objetos, e depois a **rotulação de componentes conectados** separa cada objeto em uma região distinta. Por fim, **descritores geométricos** (área e caixa delimitadora) resumem cada componente encontrado.

Ver na Figura 4.39 uma simulação deste EP.

4.9.10.1 Diretrizes de Implementação

1. **Dimensões da imagem:** ler os inteiros L (linhas) e C (colunas) de f.
2. **Dimensões de B:** ler os inteiros L_B (linhas) e C_B (colunas) do elemento estruturante.
3. **Elemento estruturante:** ler a matriz B, contendo valores 0 ou 1, linha a linha.
4. **Dados:** ler a matriz binária f (valores 0 ou 1), linha a linha.

5. **Separação:** calcular
$$f_{ero} = f \ominus B$$
usando erosão binária plana (como no EP04_04), eliminando conexões frágeis entre objetos.

6. **Rotulação:** sobre f_{ero}, identificar componentes conectados usando conectividade definida pela vizinhança B. A rotulação deve seguir varredura *raster*: ao encontrar um pixel 1 ainda não rotulado, atribuir um novo rótulo inteiro crescente a partir de 1 e propagar esse rótulo a toda a região conectada.

7. **Descritores:** para cada rótulo k, calcular:
   - **Área:** número de pixels pertencentes ao rótulo;
   - **Caixa delimitadora:**
$$(y_{min}, x_{min}, y_{max}, x_{max})$$

8. **Saída:** exibir o número total de rótulos e, em seguida, uma linha por rótulo no formato:
$$k,\ \text{área},\ y_{min},\ x_{min},\ y_{max},\ x_{max}$$

4.9.10.2 📌 Restrições Computacionais

- A erosão deve ser aplicada antes da rotulação.
- A conectividade é fixa e definida pela vizinhança acima.
- O elemento estruturante B não interfere na conectividade da rotulação.
- Sem padding em qualquer etapa.
- A ordem dos rótulos segue a primeira descoberta em varredura *raster*.

4.9.10.3 🧠 Fundamentação Teórica

Conceito	Significado	Impacto
Ponte fina	Conexão estreita entre objetos	Pode ser removida pela erosão morfológica
Conectividade	Definida pelo conjunto $\mathcal{N}(y, x)$	Determina quais pixels pertencem ao mesmo componente
Área	Número de pixels por componente	Estimativa direta do tamanho do objeto
Caixa delimitadora	Extensão espacial do rótulo	Resumo geométrico do componente

4.9.10.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: inteiro L
- Linha 2: inteiro C
- Linha 3: inteiro L_B
- Linha 4: inteiro C_B
- Próximas L_B linhas: matriz B
- Próximas L linhas: matriz f

Saída:

- Linha 1: número total de rótulos encontrados
- Linhas seguintes:
$$k,\ \text{área},\ y_{min},\ x_{min},\ y_{max},\ x_{max}$$

```
%%writefile EP04_10.cpp
// sua solução
```

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap04/sim-ep0410-rotulacao.html

Figura 4.39: Simulador EP04_10: Separação de Blobs, Rotulação e Descritores

```
Overwriting EP04_10.cpp
```

```
TestSuite("EP04_10.cpp").run()
```

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

Capítulo 5

Transformadas e Compressão

Nos capítulos anteriores, todas as operações foram realizadas **no domínio espacial**, em que os algoritmos atuam diretamente sobre os valores de intensidade dos pixels.

Neste capítulo será apresentada uma abordagem complementar: o **domínio da frequência**, no qual a imagem é representada pelas variações espaciais de intensidade, e não apenas pelos valores individuais dos pixels.

O conceito de **frequência espacial** descreve a rapidez com que a intensidade varia ao longo da imagem. Variações lentas correspondem a **baixas frequências**, enquanto bordas, detalhes finos e ruídos correspondem a **altas frequências**.

Essa representação baseia-se no fato de que qualquer imagem digital discreta pode ser decomposta em uma combinação de **funções ortogonais**. A **Transformada de Fourier** utiliza uma **base de exponenciais complexas bidimensionais** (equivalentes a senoides com orientação e frequência específicas). Outras transformadas, como a **Transformada de Cossenos (DCT)** e a **Transformada *Wavelet* (DWT)**, utilizam diferentes famílias de funções de base — cossenos bidimensionais no caso da DCT, e funções com suporte compacto no caso das *wavelets*.

Entre as principais aplicações dessa representação destacam-se:

1. **Filtragem no domínio da frequência**, para atenuar ou realçar determinadas faixas de frequência;
2. **Análise multirresolução por meio de transformadas *wavelet***, que representa estruturas em diferentes escalas;
3. **Compressão de imagens**, pela redução do número de coeficientes necessários para representar a imagem.

5.1 Objetivos

Ao concluir este capítulo, você será capaz de:

- **Interpretar o espectro de Fourier** de uma imagem, distinguindo magnitude, fase e componentes de frequência;
- **Aplicar o Teorema da Convolução** para realizar filtragem no domínio da frequência utilizando a Transformada Rápida de Fourier (FFT);
- **Projetar e analisar filtros no domínio da frequência**, compreendendo o funcionamento de filtros passa-baixa, passa-alta e *notch*;
- **Compreender a análise multirresolução por transformadas *wavelet*** e sua aplicação na representação hierárquica de imagens;
- **Descrever o processo de compressão de imagens**, incluindo a Transformada Discreta do Cosseno (DCT) e a quantização dos coeficientes;
- **Selecionar formatos de armazenamento de imagens**, como JPEG, PNG e WebP, de acordo com os requisitos da aplicação.

5.2 Configuração do Ambiente

```
import os, urllib.request

os.makedirs("tmp/state", exist_ok=True)  # artefatos de build da trilha C++

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")

# Kernel Python mesmo na trilha C++. cpp=True baixa morph.hpp + stb; as
# células %%writefile *.cpp deste capítulo compilam COM OpenCV
# (-DMM_USE_OPENCV + pkg-config opencv4).
import config
config.setup(cpp=True)
from morph import mm
import numpy as np
```

```
Ambiente pronto. Morph: 1.1.9 | OpenCV: 5.0.0
```

5.3 Transformada de Fourier Discreta 2D

A análise de Fourier baseia-se no princípio de que qualquer sinal periódico pode ser representado como uma soma de funções senoidais com diferentes frequências, amplitudes e fases. Esse conceito também se aplica às imagens digitais, permitindo representá-las no **domínio da frequência** em vez do domínio espacial.

A Figura 5.1 ilustra essa decomposição para um sinal unidimensional. No caso de uma imagem, a Transformada Discreta de Fourier (DFT) converte a matriz de intensidades $f(x, y)$ em um conjunto de coeficientes que descreve a contribuição das diferentes frequências espaciais presentes na imagem.

```
%%writefile tmp/fig_decomposicao_1d.cpp
#define MM_OUT "tmp/fig_decomposicao_1d.png"
//| label: fig-decomposicao-1d
//| fig-cap: "Decomposição de Fourier 1D: uma onda quadrada (linha tracejada) é aproximada
pela soma das primeiras senoides (linhas coloridas). Quanto mais termos, melhor a
aproximação."
//| echo: false
//| output: true

#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <cmath>
#include "morph.hpp"
#include <filesystem>

int main() {
    // Gera pontos x de 0 a 2*pi
    std::vector<double> x(400);
    for (int i = 0; i < 400; ++i) {
        x[i] = 2.0 * M_PI * i / 399.0;
    }

    // Onda quadrada ideal (1.0 para x < pi, -1.0 caso contrário)
    std::vector<double> square(400);
    for (int i = 0; i < 400; ++i) {
        square[i] = (x[i] < M_PI) ? 1.0 : -1.0;
    }
```

```
    // Soma das primeiras harmônicas
    std::vector<double> soma(400, 0.0);
    std::vector<std::vector<double>> harms;
    for (int n = 1; n <= 3; ++n) {
        std::vector<double> h(400);
        double coef = (4.0 / M_PI) * (1.0 / (2 * n - 1));
        for (int i = 0; i < 400; ++i) {
            h[i] = coef * std::sin((2 * n - 1) * x[i]);
        }
        harms.push_back(h);
        for (int i = 0; i < 400; ++i) {
            soma[i] += h[i];
        }
    }

    // Prepara as curvas para o gráfico
    std::vector<std::vector<double>> ys = {square, harms[0], harms[1], harms[2], soma};
    std::vector<std::string> labels = {"Onda quadrada ideal", "1a harmonica", "3a harmonica",
"5a harmonica", "Soma (3 primeiras)"};
    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)};

    // Cria o gráfico de linha (ordem: x único, ys, labels, colors, title, xlabel, ylabel)
    mm::Image chart = mm::lineChart(
        x, ys, labels, colors,
        "Sintese de Fourier: de senos a uma onda quadrada",
        "Posicao", "Intensidade"
    );

    // Exibe resultado
    std::vector<mm::Image> chart_list = {chart};
    mm::show(chart_list, MM_OUT, {"Decomposicao de Fourier 1D"}, 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] Decomposicao de Fourier 1D
```

```
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)")
```

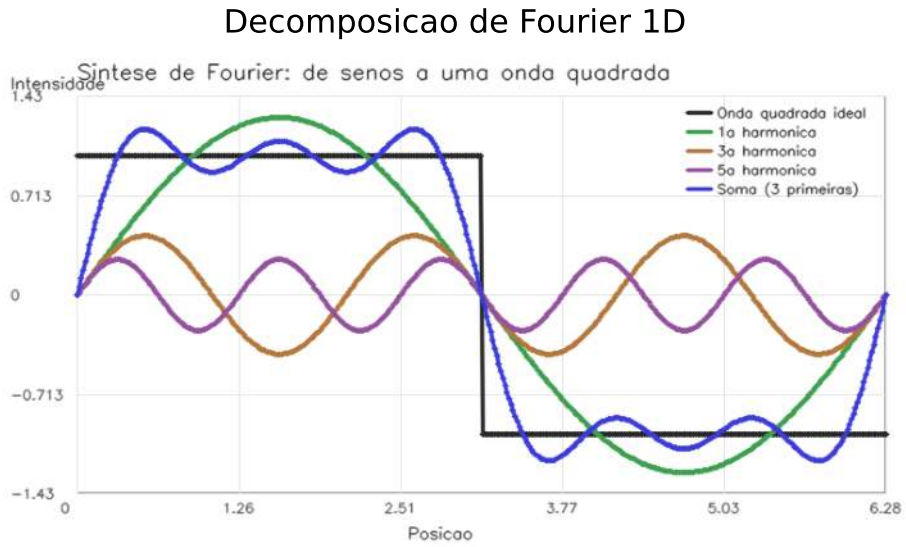


Figura 5.1: Decomposição de Fourier 1D: uma onda quadrada (linha tracejada) é aproximada pela soma das primeiras senoides (linhas coloridas). Quanto mais termos, melhor a aproximação.

5.3.1 Simulador: Reconstruindo Sinais com Senoides

Antes de estudar imagens bidimensionais, o simulador da Figura 5.2 ilustra o princípio da análise de Fourier para sinais unidimensionais: **uma forma de onda pode ser aproximada pela soma de senoides com diferentes frequências e amplitudes**.

À medida que novos termos são adicionados, a soma das senoides (curva preta) aproxima-se da forma de onda de referência (tracejada). O gráfico inferior apresenta o espectro de amplitudes, indicando a contribuição de cada frequência para a reconstrução do sinal.

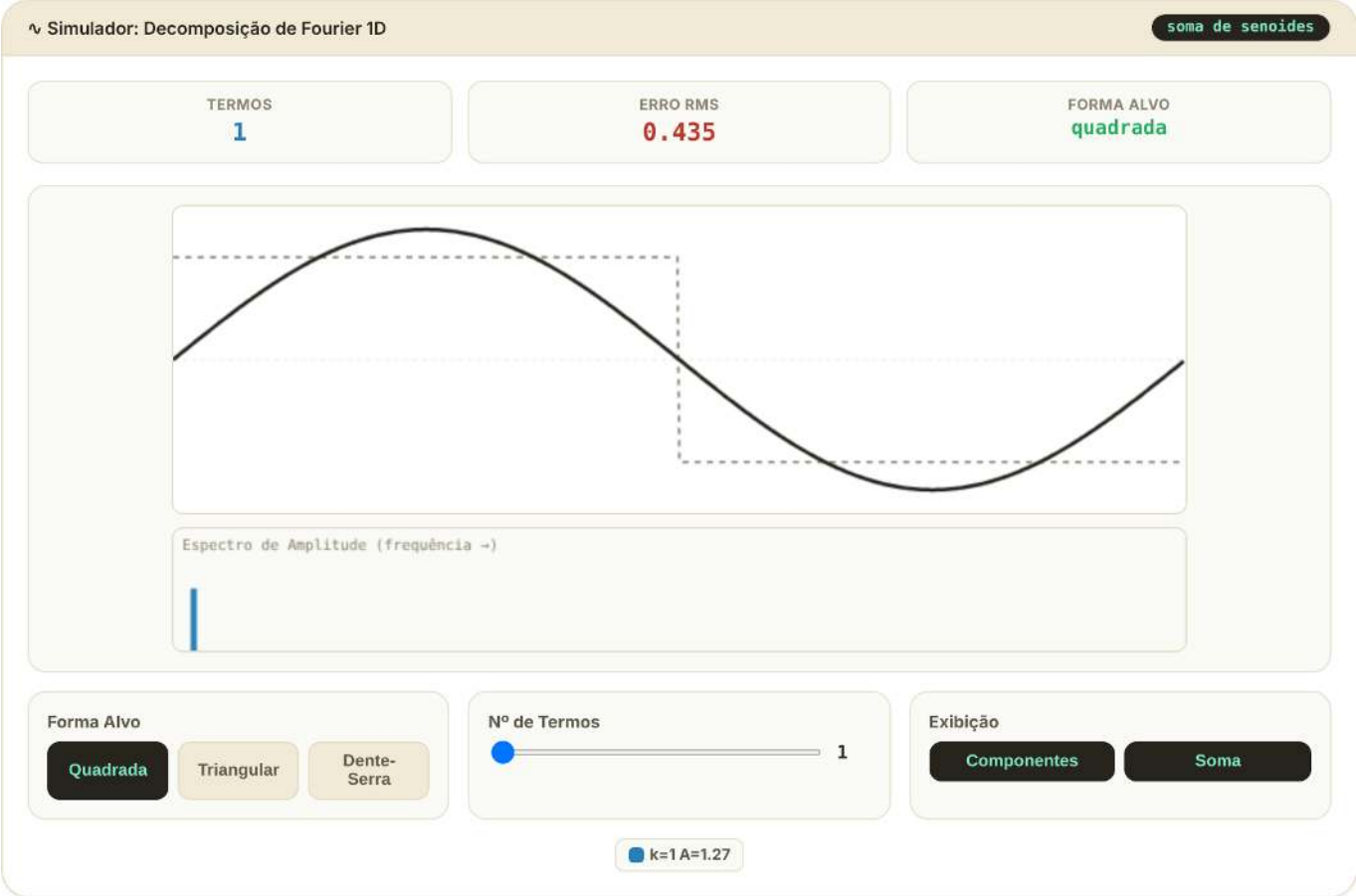
Atividade

Explore o simulador e responda:

1. Quantos termos são necessários para obter uma boa aproximação da onda quadrada?
2. Qual das três formas de onda converge mais rapidamente? Justifique sua resposta.
3. Como o espectro de amplitudes se altera ao trocar a onda quadrada pela triangular?

Respostas

1. Quantos termos são necessários para uma boa aproximação da onda quadrada?
Com aproximadamente 15 a 20 termos, a forma da onda já se aproxima bem da referência. Entretanto, próximo às descontinuidades permanece uma pequena oscilação, conhecida como **fenômeno de Gibbs**, que não desaparece mesmo com a adição de mais termos.
2. Qual forma converge mais rapidamente? Por quê?
A **onda triangular** converge mais rapidamente, pois as amplitudes de seus harmônicos decaem mais rápido que as da onda quadrada e da onda dente-de-serra. Como consequência, poucos termos já produzem uma boa aproximação.
3. Como o espectro muda entre a onda quadrada e a triangular?
Ambas possuem apenas **harmônicos ímpares**, mas, na onda triangular, as amplitudes diminuem muito mais rapidamente. Assim, poucos harmônicos são suficientes para reconstruir o sinal com boa precisão.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap05/sim-05-freq.html

Figura 5.2: Simulador interativo da decomposição de Fourier 1D: visualização da soma de senoides com diferentes frequências, amplitudes e fases. Adicione termos e observe a convergência para formas de onda arbitrárias.

5.3.2 Interpretação do espectro de frequência

Ao aplicar a Transformada Discreta de Fourier (DFT) a uma imagem e visualizar o módulo de seus coeficientes (ver Figura 5.5), obtém-se o **espectro de magnitude**, que mostra a distribuição das frequências espaciais presentes na imagem.

O coeficiente localizado na origem da DFT, denominado **componente DC** (*Direct Current*), corresponde à frequência nula e representa a intensidade média da imagem. Por convenção, esse coeficiente é armazenado

no canto superior esquerdo do espectro. Para facilitar sua interpretação, aplica-se a operação **FFT Shift**, que desloca a componente DC para o centro da imagem. Após esse deslocamento, as baixas frequências concentram-se na região central, enquanto as altas frequências ficam próximas às bordas, como resume a Tabela 5.1.

Tabela 5.1: Correspondência entre as regiões do espectro de magnitude após a aplicação do FFT Shift.

Região do espectro	Componentes predominantes	Exemplos na imagem
Centro (baixas frequências)	Variações espaciais lentas	Iluminação, regiões homogêneas e formas globais
Região intermediária (médias frequências)	Variações de escala intermediária	Texturas e padrões repetitivos
Bordas (altas frequências)	Variações espaciais rápidas	Contornos, detalhes finos e ruído

Essa organização facilita a interpretação do espectro e o projeto de filtros. A atenuação das baixas frequências reduz as variações globais de intensidade, enquanto a atenuação das altas frequências suaviza a imagem ao reduzir detalhes finos e parte do ruído.

5.3.3 O Experimento da Grade: Construindo uma Imagem a partir de um Único Coeficiente

Antes de apresentar a formulação matemática da Transformada Discreta de Fourier (DFT), é útil analisar sua inversa, denominada Transformada Discreta Inversa de Fourier (IDFT). Considere um espectro em que todos os coeficientes sejam nulos, exceto um. Um exemplo dessa construção é apresentado no código da Figura 5.3 e pode ser explorado interativamente no simulador da Figura 5.4.

A imagem reconstruída é uma senoide bidimensional. A posição do coeficiente no espectro determina sua **orientação** e sua **frequência espacial**, enquanto sua magnitude e sua fase definem, respectivamente, sua amplitude e seu deslocamento espacial. Assim, cada coeficiente da DFT representa uma componente senoidal, e a imagem original pode ser reconstruída pela soma de todas essas componentes.

```
%%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 <complex>
#include <cmath>
#include "morph.hpp"
#include <filesystem>

// Helper para fftshift manual em matriz complexa
void fftshift_complex(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, cx, cy));  // Top-Right
    cv::Mat q2(src, cv::Rect(0, cy, cx, cy));  // Bottom-Left
    cv::Mat q3(src, cv::Rect(cx, cy, cx, cy)); // Bottom-Right
    cv::Mat tmp;
    cv::hconcat(q3, q2, tmp);
    cv::hconcat(tmp, q0, tmp);
```

```
    cv::hconcat(tmp, q1, dst); // Reorganiza em 4 quadrantes trocados
}

// Helper para calcular magnitude e normalizar
void normalize_float_mat(cv::Mat& src, cv::Mat& dst) {
    cv::normalize(src, dst, 0, 255, cv::NORM_MINMAX, CV_8U);
}

int main() {
    int N_grid = 100;

    // Espectro complexo vazio
    cv::Mat espectro_vazio(N_grid, N_grid, CV_64FC2, cv::Scalar(0, 0));

    // Acendendo um único ponto (frequência) fora do centro
    int u0 = 10, v0 = 5;
    int idx_y = N_grid/2 - v0;
    int idx_x = N_grid/2 - u0;
    espectro_vazio.at<cv::Vec2d>(idx_y, idx_x) = cv::Vec2d(1000, 0);

    // FFT shift manual (para alinhar com o domínio espacial)
    cv::Mat espectro_shifted;
    fftshift_complex(espectro_vazio, espectro_shifted);

    // Retornando para o domínio espacial (IDFT)
    cv::Mat onda_2d_complex;
    cv::idft(espectro_shifted, onda_2d_complex, cv::DFT_SCALE);

    // Extrair parte real
    cv::Mat onda_2d_channels[2];
    cv::split(onda_2d_complex, onda_2d_channels);
    cv::Mat onda_2d = onda_2d_channels[0]; // Parte real

    // Normalizar para visualização
    cv::Mat onda_vis;
    normalize_float_mat(onda_2d, onda_vis);

    // |Especro| para visualização
    cv::Mat magnitude;
    cv::Mat espectro_channels[2];
    cv::split(espectro_vazio, espectro_channels);
    cv::magnitude(espectro_channels[0], espectro_channels[1], magnitude);

    cv::Mat espectro_vis;
    normalize_float_mat(magnitude, espectro_vis);

    // Destaque visual do ponto (converter para BGR e desenhar círculo)
    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);

    // Converter para mm::Image e exibir
    mm::Image img_espectro(espectro_color.rows, espectro_color.cols,
espectro_color.channels());
    std::memcpy(img_espectro.data.data(), espectro_color.data, img_espectro.data.size());

    mm::Image img_onda(onda_vis.rows, onda_vis.cols, onda_vis.channels());
    std::memcpy(img_onda.data.data(), onda_vis.data, img_onda.data.size());
```

```
    std::vector<mm::Image> imagens = {img_espectro, img_onda};
    std::vector<std::string> titulos = {"Espectro (1 ponto ativo)", "Onda 2D Resultante
(IDFT)"};

    mm::show(imagens, MM_OUT, titulos, 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)")
```

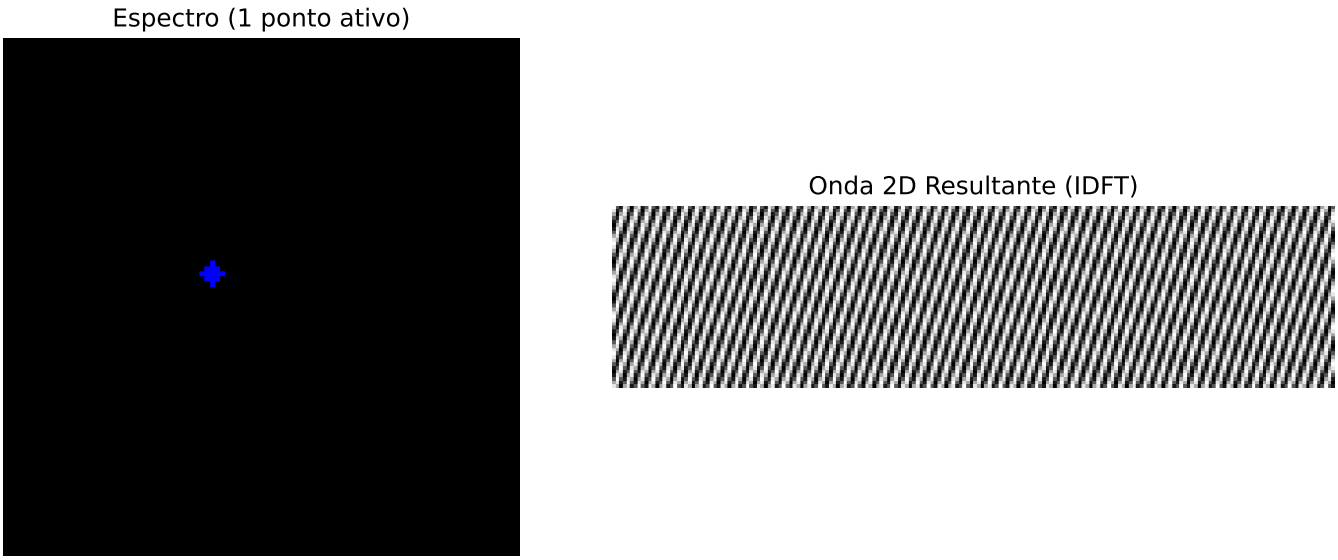
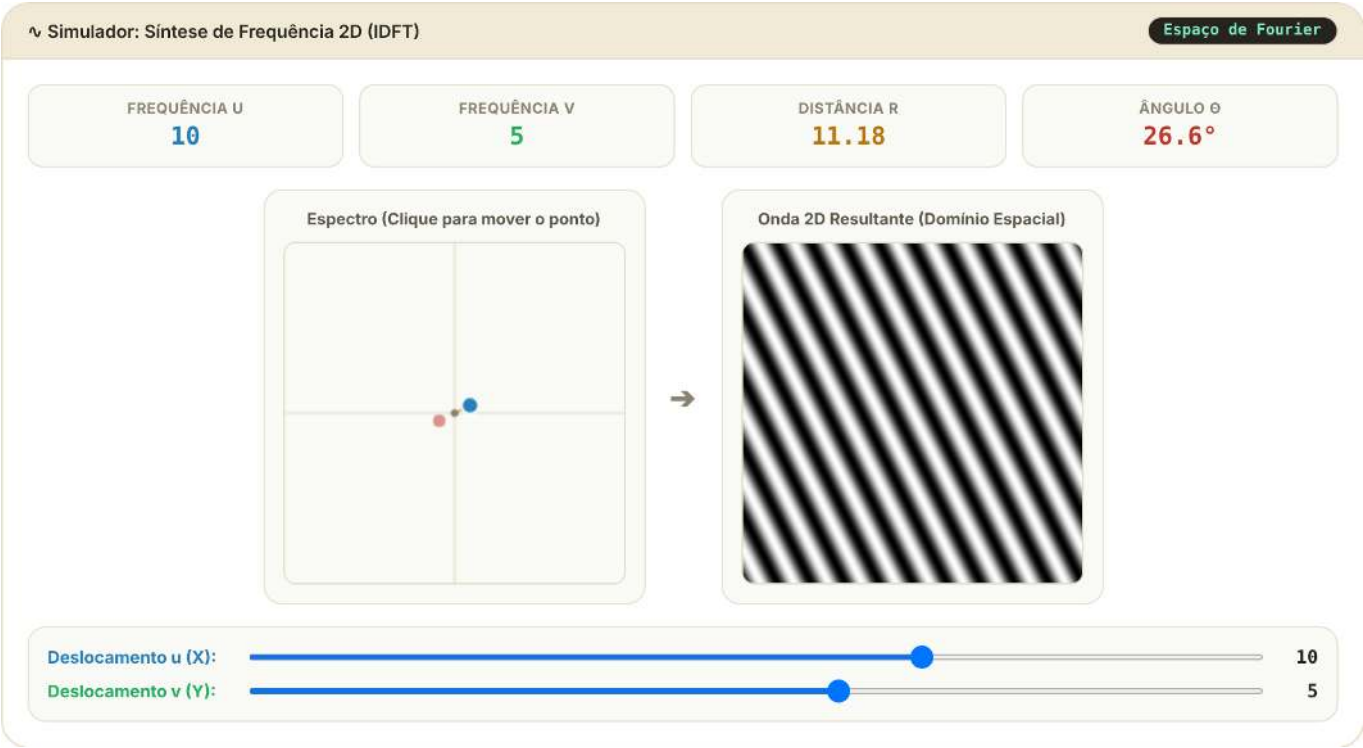


Figura 5.3: Toda frequência no espectro (ponto isolado) corresponde a uma onda senoidal 2D rotacionada no domínio espacial.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap05/sim-05-grade-2d.html

Figura 5.4: Simulador interativo da síntese de Fourier 2D. Altere a posição horizontal (u) e vertical (v) do coeficiente no espectro de frequências centrado e observe como a distância em relação ao centro dita a frequência espacial (espessura) e o ângulo dita a orientação da onda senoidal gerada.

5.3.4 Definição Matemática

Considere uma imagem $f(x, y)$ com dimensões $M \times N$. Sua **Transformada Discreta de Fourier 2D** (DFT) é definida por:

$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y)\, e^{-j2\pi\left(\frac{ux}{M} + \frac{vy}{N}\right)} \tag{5.1}$$

em que $u = 0, 1, \ldots, M-1$ e $v = 0, 1, \ldots, N-1$ representam as frequências discretas nas direções horizontal e vertical, respectivamente. O termo exponencial corresponde a uma senoide bidimensional, cuja frequência e orientação são determinadas pelos índices (u, v).

A **Transformada Discreta Inversa de Fourier 2D** (IDFT) reconstrói a imagem original a partir de seus coeficientes:

$$f(x,y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v)\, e^{j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)} \tag{5.2}$$

As Equações Equação 5.1 e Equação 5.2 mostram que a DFT e a IDFT formam um par de transformações: a primeira converte a imagem para o domínio da frequência, enquanto a segunda reconstrói exatamente a imagem original a partir de seus coeficientes.

i Sobre o símbolo j

O termo j denota a **unidade imaginária**, definida por $j^2 = -1$. Em engenharia e processamento de sinais, adota-se j em vez de i para evitar conflito com a notação de corrente elétrica. Sua utilização na exponencial complexa, regida pela fórmula de Euler ($e^{j\theta} = \cos\theta + j\sin\theta$), permite representar de forma compacta a amplitude e a fase de cada frequência espacial presente na imagem.

i O que é o componente DC?

O coeficiente $F(0,0)$, denominado **componente DC** (*Direct Current*), é igual à soma das intensidades de todos os pixels da imagem (ver Figura 5.5):

$$F(0,0) = MN\,\bar{f},$$

em que $\bar{f}$ é a intensidade média da imagem. Por isso, o componente DC representa o nível médio de intensidade e, na maioria das imagens naturais, possui a maior magnitude do espectro.
Os demais coeficientes representam variações em torno dessa média. Após a aplicação do **FFT Shift**, o componente DC é deslocado para o centro do espectro, concentrando as baixas frequências na região central e as altas frequências nas bordas.

5.3.5 Magnitude e Fase

Cada coeficiente da Transformada Discreta de Fourier (DFT) é um número complexo e pode ser escrito como

$$F(u,v) = R(u,v) + j\,I(u,v),$$

em que $R(u,v)$ e $I(u,v)$ correspondem, respectivamente, às partes **real** e **imaginária** do coeficiente. Da Equação Equação 5.1, obtêm-se

$$R(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \cos\left(2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)\right),$$

e

$$I(u,v) = -\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \sin\left(2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)\right).$$

A partir dessa representação, definem-se duas grandezas fundamentais:

Figura 5.5: Diagrama conceitual do espectro de Fourier 2D centrado.

- **Magnitude**, que indica a intensidade da componente de frequência,

$$|F(u,v)| = \sqrt{R(u,v)^2 + I(u,v)^2};$$

- **Fase**, que determina o alinhamento (ou deslocamento) espacial da componente,

$$\phi(u,v) = \operatorname{atan2}(I(u,v),\, R(u,v))\,.$$

Assim, cada coeficiente também pode ser escrito em sua forma polar,

$$F(u,v) = |F(u,v)|\, e^{j\phi(u,v)}.$$

O espectro de Fourier pode, portanto, ser visualizado por meio de duas imagens distintas: o **espectro de magnitude**, normalmente utilizado para analisar a distribuição das frequências, e o **espectro de fase**, que descreve a organização espacial das componentes senoidais.

Embora o espectro de magnitude seja o mais utilizado para inspeção visual, a fase contém grande parte das informações estruturais da imagem. A combinação de magnitude e fase permite reconstruir exatamente a imagem original por meio da IDFT.

5.3.6 O que a Magnitude e a Fase carregam?

Uma demonstração clássica consiste em combinar a magnitude de uma imagem com a fase de outra e reconstruir o resultado. Esse experimento evidencia que:

- **A fase** preserva a estrutura espacial da imagem, incluindo a posição dos objetos, seus contornos e sua geometria. Pequenas alterações na fase podem provocar grandes mudanças visuais.
- **A magnitude** controla como a energia é distribuída entre as frequências espaciais, influenciando principalmente o contraste e a textura.

Quando uma imagem é reconstruída com a magnitude de A e a fase de B, o resultado tende a **assemelhar-se mais a B do que a A**, evidenciando que a fase é o principal componente responsável pela organização

espacial da cena. Entretanto, a magnitude continua sendo importante, pois modula o contraste das estruturas reconstruídas. Assim, uma reconstrução fiel depende da combinação consistente entre magnitude e fase.

Um exemplo desse comportamento é apresentado na Figura 5.6.

i Analogia com Áudio: Limitações e Cuidados

A fase de um sinal desempenha papéis distintos em áudio e imagens:

- **Áudio estéreo ou multicanal:** a fase relativa entre os canais é fundamental para a percepção da posição das fontes sonoras, por meio das diferenças interaurais de tempo (ITD, *Interaural Time Differences*).
- **Áudio monaural:** a fase absoluta exerce pouca influência perceptual direta.
- **Imagens (DFT):** a fase é o principal fator responsável pela organização espacial da cena, enquanto a magnitude modula o contraste e a distribuição da energia entre as frequências.

Em ambos os domínios, a **magnitude** está relacionada à intensidade das componentes de frequência: em áudio, influencia o timbre e a intensidade percebida; em imagens, influencia o contraste e a textura.

```
%%writefile tmp/fig_05_dft_intro.cpp
#define MM_OUT "tmp/fig_05_dft_intro.png"
#include <opencv2/opencv.hpp>
#include <iostream>
#include <vector>
#include <cmath>
#include <numeric>
#include "morph.hpp"
#include <filesystem>

//| label: fig-05-dft-intro
//| fig-cap: "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."
//| echo: true
//| output: true

// Função auxiliar para centralizar o espectro (fftshift)
void fftShift(cv::Mat& spectrum) {
    // Divide o espectro em 4 quadrantes e troca suas posições
    int cx = spectrum.cols / 2;
    int cy = spectrum.rows / 2;

    // Cria regiões para os 4 quadrantes
    cv::Mat q0(spectrum, cv::Rect(0, 0, cx, cy));   // Topo-esquerda
    cv::Mat q1(spectrum, cv::Rect(cx, 0, cx, cy));  // Topo-direita
    cv::Mat q2(spectrum, cv::Rect(0, cy, cx, cy));  // Baixo-esquerda
    cv::Mat q3(spectrum, cv::Rect(cx, cy, cx, cy)); // Baixo-direita

    // Troca os quadrantes
    cv::Mat tmp;
    q0.copyTo(tmp);
    q3.copyTo(q0);
    tmp.copyTo(q3);

    q1.copyTo(tmp);
    q2.copyTo(q1);
    tmp.copyTo(q2);
```

```
}

// Função para conversão cv::Mat para mm::Image
mm::Image cvToMmImage(const cv::Mat& mat) {
    cv::Mat converted;
    if (mat.channels() == 1) {
        // Converte para formato adequado
        cv::Mat normalized;
        cv::normalize(mat, normalized, 0, 255, cv::NORM_MINMAX, CV_8U);
        mm::Image img(normalized.rows, normalized.cols, 1);
        std::memcpy(img.data.data(), normalized.data, img.data.size());
        return img;
    } else {
        cv::Mat bgr;
        cv::cvtColor(mat, bgr, cv::COLOR_BGR2RGB);
        mm::Image img(bgr.rows, bgr.cols, 3);
        std::memcpy(img.data.data(), bgr.data, img.data.size());
        return img;
    }
}

int main() {
    //   Experimento: A Importância da Fase
    //   Carregamento da imagem
    std::string url     =
"https://upload.wikimedia.org/wikipedia/commons/2/25/GAZI.MD.AHAD_11.jpg";
    std::string caminho = "imagens/coins.jpg";

    mm::Image img_obj;

    // Verifica se o arquivo existe (simplificado - sempre tenta ler a URL ou usa o caminho)
    // Em C++, verificamos se o arquivo pode ser aberto
    FILE* file = fopen(caminho.c_str(), "rb");
    if (file) {
        fclose(file);
        img_obj = mm::read(caminho);
    } else {
        // Tenta criar diretório e baixar
        std::system("mkdir -p imagens");
        img_obj = mm::read(url);
        mm::write(img_obj, caminho);
    }

    // Converte mm::Image para cv::Mat
    cv::Mat img_color(img_obj.h, img_obj.w, CV_8UC3, img_obj.data.data());

    // Converte para cinza
    cv::Mat img_gray_cv;
    cv::cvtColor(img_color, img_gray_cv, cv::COLOR_RGB2GRAY);

    // Redimensiona para 400x400
    cv::Mat img_a;
    cv::resize(img_gray_cv, img_a, cv::Size(400, 400));

    // Cria uma imagem B sintética (padrão geométrico)
    cv::Mat img_b = cv::Mat::zeros(400, 400, CV_8U);
    cv::rectangle(img_b, cv::Point(100, 100), cv::Point(300, 300), cv::Scalar(255), -1);
    cv::circle(img_b, cv::Point(200, 200), 150, cv::Scalar(128), 10);

    // Converte para float para FFT
    cv::Mat img_a_float, img_b_float;
```

```
    img_a.convertTo(img_a_float, CV_32F);
    img_b.convertTo(img_b_float, CV_32F);

    // FFT de A e B
    cv::Mat FA_complex, FB_complex;
    cv::dft(img_a_float, FA_complex, cv::DFT_COMPLEX_OUTPUT);
    cv::dft(img_b_float, FB_complex, cv::DFT_COMPLEX_OUTPUT);

    // Separa canais reais e imaginários
    cv::Mat FA_real[2], FA_imag[2];
    cv::split(FA_complex, FA_real);
    FA_imag[0] = FA_real[0].clone();
    FA_imag[1] = FA_real[1].clone();

    cv::Mat FB_real[2], FB_imag[2];
    cv::split(FB_complex, FB_real);
    FB_imag[0] = FB_real[0].clone();
    FB_imag[1] = FB_real[1].clone();

    // Calcula magnitude e fase de A
    cv::Mat magA, phaseA;
    cv::cartToPolar(FA_real[0], FA_real[1], magA, phaseA);

    // Calcula magnitude e fase de B
    cv::Mat magB, phaseB;
    cv::cartToPolar(FB_real[0], FB_real[1], magB, phaseB);

    // Experimento 1: Magnitude de A, Fase de B
    cv::Mat recA_complex_real, recA_complex_imag;
    cv::polarToCart(magA, phaseB, recA_complex_real, recA_complex_imag);

    cv::Mat recA_complex[2] = {recA_complex_real, recA_complex_imag};
    cv::Mat recA_spectrum;
    cv::merge(recA_complex, 2, recA_spectrum);

    // IFFT para recuperar imagem
    cv::Mat rec_A_mag_B_fase_float;
    cv::idft(recA_spectrum, rec_A_mag_B_fase_float, cv::DFT_SCALE | cv::DFT_REAL_OUTPUT);

    // Normaliza e converte para uint8
    cv::Mat rec_A_mag_B_fase;
    cv::normalize(rec_A_mag_B_fase_float, rec_A_mag_B_fase, 0, 255, cv::NORM_MINMAX, CV_8U);

    // Experimento 2: Magnitude de B, Fase de A
    cv::Mat recB_complex_real, recB_complex_imag;
    cv::polarToCart(magB, phaseA, recB_complex_real, recB_complex_imag);

    cv::Mat recB_complex[2] = {recB_complex_real, recB_complex_imag};
    cv::Mat recB_spectrum;
    cv::merge(recB_complex, 2, recB_spectrum);

    // IFFT para recuperar imagem
    cv::Mat rec_B_mag_A_fase_float;
    cv::idft(recB_spectrum, rec_B_mag_A_fase_float, cv::DFT_SCALE | cv::DFT_REAL_OUTPUT);

    // Normaliza e converte para uint8
    cv::Mat rec_B_mag_A_fase;
    cv::normalize(rec_B_mag_A_fase_float, rec_B_mag_A_fase, 0, 255, cv::NORM_MINMAX, CV_8U);

    // Converte todos para mm::Image para exibição
    mm::Image img_a_mm = cvToMmImage(img_a);
```

```
    mm::Image img_b_mm = cvToMmImage(img_b);
    mm::Image rec_A_mm = cvToMmImage(rec_A_mag_B_fase);
    mm::Image rec_B_mm = cvToMmImage(rec_B_mag_A_fase);

    // Exibe as imagens
    std::vector<mm::Image> images = {img_a_mm, img_b_mm, rec_A_mm, rec_B_mm};
    std::vector<std::string> titles = {"Imagem A", "Imagem B", "Mag(A) + Fase(B)", "Mag(B) +
Fase(A)"};
    mm::show(images, MM_OUT, titles, 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;

    // Mantém img_gray (a versão em cinza da imagem original) no escopo
    mm::Image img_gray = cvToMmImage(img_gray_cv);


// [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)")
```

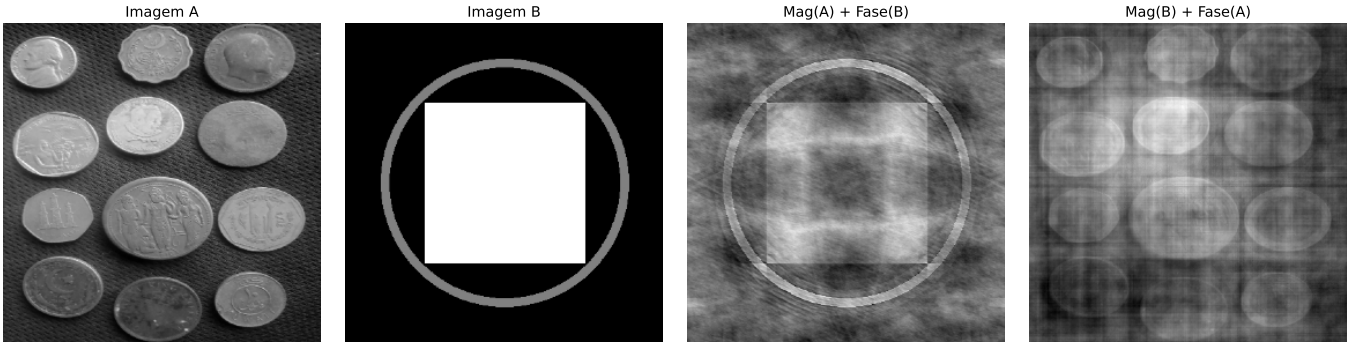


Figura 5.6: Experimento de troca de fase: Imagem A (moedas) e Imagem B (padrão geométrico) reconstruídas com magnitudes e fases trocadas. O resultado mostra que a estrutura visual é **muito mais sensível à fase** do que à magnitude: quando a fase de B é mantida, a imagem resultante preserva a organização 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 Teorema da Convolução e Estratégias de Filtragem

O **Teorema da Convolução** estabelece uma relação fundamental entre os domínios espacial e da frequência:

$$f(x,y) \circledast h(x,y) \overset{\mathcal{F}}{\longleftrightarrow} F(u,v)\,H(u,v) \tag{5.3}$$

em que $\circledast$ representa a **convolução circular discreta**. Assim, a convolução entre uma imagem $f(x,y)$ e um filtro $h(x,y)$ pode ser substituída pela multiplicação de seus espectros.

Na prática, para obter o mesmo resultado da convolução linear realizada no domínio espacial, aplica-se **zero-padding** antes da Transformada Rápida de Fourier (FFT), evitando artefatos nas bordas da imagem.

Entretanto, nem sempre a filtragem no domínio da frequência é a alternativa mais eficiente. Para filtros como o Gaussiano e o filtro da média (*Box Filter*), a propriedade de **separabilidade** permite reduzir significativamente o custo computacional da convolução no domínio espacial.

5.4.1 *Kernel* Separável vs. Não Separável

Um ***kernel* separável** pode ser escrito como o produto externo de dois vetores unidimensionais,

$$H = v\, h^T,$$

permitindo que a convolução bidimensional seja substituída por duas convoluções unidimensionais consecutivas: uma na direção horizontal e outra na vertical.

Já um ***kernel* não separável** não admite essa decomposição e, portanto, sua convolução deve ser realizada diretamente sobre a vizinhança bidimensional.

Na prática, para um *kernel* de dimensão $K \times K$, a convolução direta exige K^2 multiplicações por pixel, enquanto um *kernel* separável requer apenas $2K$ multiplicações, reduzindo significativamente o custo computacional.

5.4.2 Análise de Eficiência Computacional

Considere uma imagem de dimensões $M \times N$ e um filtro quadrado de tamanho $K \times K$. A Tabela 5.2 compara a complexidade das principais estratégias de filtragem.

Tabela 5.2: Comparação da complexidade da convolução direta, separável e via Transformada Rápida de Fourier (FFT).

Método de Filtragem	Complexidade Assintótica	Dependência de K	Aplicação típica
Espacial não separável	$\mathcal{O}(MNK^2)$	Quadrática	*Kernels* pequenos e não separáveis
Espacial separável	$\mathcal{O}(MNK)$	Linear	Filtros Gaussiano e da média
Via FFT	$\mathcal{O}(MN\log(MN))$	Independente de K	*Kernels* grandes

Para *kernels* pequenos, a convolução espacial, especialmente quando o filtro é separável, costuma ser mais eficiente devido ao baixo custo das operações. À medida que o tamanho do *kernel* aumenta, a filtragem via FFT torna-se mais vantajosa, pois seu custo praticamente independe da dimensão do filtro.

5.4.3 Discussão dos resultados experimentais

O gráfico obtido no ensaio com a imagem das moedas (2560×1920), apresentado na Figura 5.7, confirma o comportamento previsto pela análise de complexidade computacional.

1. **Convolução não separável ($\mathcal{O}(MNK^2)$)**
   A convolução direta apresenta crescimento quadrático com o tamanho do *kernel*. Para valores pequenos de K, o custo é baixo, mas aumenta rapidamente à medida que o *kernel* cresce, tornando-se inviável para aplicações em tempo real.

2. **Filtragem via FFT ($\mathcal{O}(MN\log(MN))$)**
   O custo da FFT depende apenas do tamanho da imagem, sendo independente de K. Por isso, seu desempenho permanece aproximadamente constante ao variar o *kernel*, tornando-a vantajosa para filtros grandes ou não separáveis.

3. **Convolução separável ($\mathcal{O}(MNK)$)**
   A decomposição do *kernel* em dois filtros unidimensionais reduz significativamente o custo computacional. Na prática, essa abordagem tende a ser a mais eficiente para filtros separáveis, especialmente em implementações otimizadas.

Em geral, a escolha do método depende do tamanho e da estrutura do *kernel*. Filtros separáveis são mais eficientes no domínio espacial, enquanto a FFT se torna mais vantajosa para *kernels* grandes ou múltiplas convoluções no domínio da frequência.

$$g = \mathcal{F}^{-1}[\mathcal{F}(f) \cdot \mathcal{F}(h)] \quad \text{(FFT)} \qquad g = f \circledast h \quad \text{(convolução direta)} \qquad g = (f \circledast v) \circledast h^T \quad \text{(separável)} \tag{5.4}$$

onde:

- $f(x, y)$ representa a imagem de entrada;
- $h(x, y)$ é o *kernel* bidimensional do filtro;
- v e h^T são, respectivamente, os vetores vertical e horizontal que compõem o *kernel* separável.

```
%%writefile tmp/fig_05_conv_eficiencia.cpp
#define MM_OUT "tmp/fig_05_conv_eficiencia.png"
#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <cmath>
#include "morph.hpp"
#include <filesystem>

int main() {
    //| label: fig-05-conv-eficiencia
    //| fig-cap: "Comparação de eficiência: Convolução Não Separável (Espacial 2D), Separável
    (Espacial 1D) e via FFT."
    //| echo: false
    //| output: true

    // Trilha C++: curvas de CUSTO relativo (contagem de operações) - a forma das
    // curvas (K^2 vs 2K vs log N) e o que importa; a trilha py mede tempos reais.
    std::vector<double> K = {3, 7, 11, 15, 21, 31, 41, 51};
    double N2 = 256.0 * 256.0;

    std::vector<double> t_nao_sep, t_sep;
    for (double k : K) {
        t_nao_sep.push_back(N2 * k * k / 1e6);
        t_sep.push_back(N2 * 2.0 * k / 1e6);
    }

    std::vector<double> t_fft(K.size());
    double logN2 = std::log2(N2);
    for (size_t i = 0; i < K.size(); ++i) {
        t_fft[i] = N2 * logN2 / 1e6;
    }

    // Vetores de vetores para múltiplas curvas (uma por posição)
    std::vector<std::vector<double>> xs = {K, K, K};
    std::vector<std::vector<double>> ys = {t_nao_sep, t_sep, t_fft};

    mm::Image chart = mm::lineChart(
        xs, ys,
        {"Nao Separavel  O(N^2 K^2)", "Separavel  O(N^2 . 2K)", "Via FFT  O(N^2 log N)"},
        {},
        "Custo relativo: Espacial vs Frequencia",
        "Tamanho do kernel  K",
        "Operacoes  (x10^6)"
    );

    std::vector<mm::Image> charts = {chart};
```

```
    std::vector<std::string> titles = {"Comparacao de complexidade"};
    mm::show(charts, 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_conv_eficiencia_0.png");
// [pdi:panel-io:end]
return 0;
}
```

```
Overwriting tmp/fig_05_conv_eficiencia.cpp
```

```
!g++ -I. -std=c++17 -DMM_USE_OPENCV -I/usr/include/opencv4 tmp/fig_05_conv_eficiencia.cpp -o
tmp/fig_05_conv_eficiencia -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_eficiencia \
  && test -f "tmp/fig_05_conv_eficiencia.png" \
  || echo "  mm::show não gravou tmp/fig_05_conv_eficiencia.png"
```

```
[1] Comparacao de complexidade
```

```
try:
    mm.show(
        [
            mm.read("tmp/fig_05_conv_eficiencia_0.png"),
        ],
        titles=[
            'Comparacao de complexidade',
        ],
        cols=1,
    )
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_conv_eficiencia_0.png (ver a versao Python)")
```

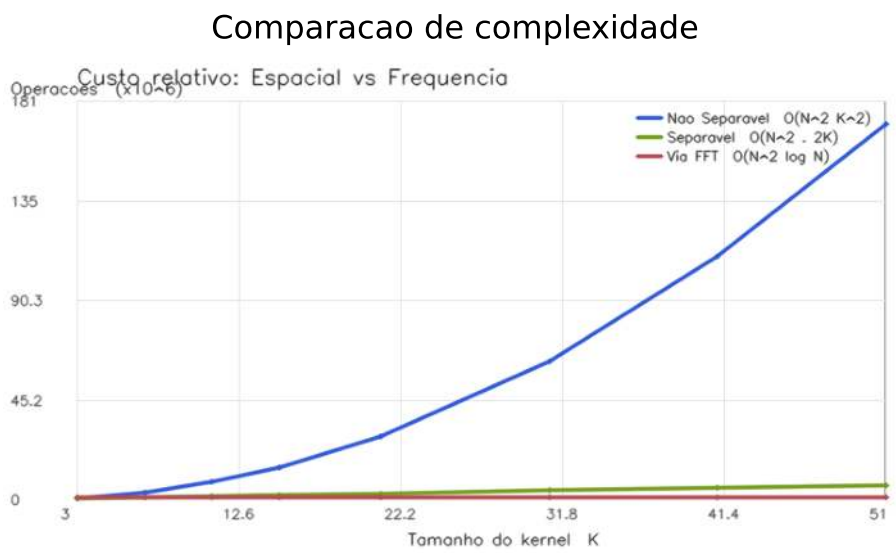


Figura 5.7: Comparação de eficiência: Convolução Não Separável (Espacial 2D), Separável (Espacial 1D) e via FFT.

! O problema da convolução circular (*wrap-around*)

A Transformada Discreta de Fourier (DFT) assume que a imagem é **periodicamente estendida no espaço**, isto é, que suas bordas se repetem indefinidamente.
Nessa condição, a multiplicação no domínio da frequência corresponde a uma **convolução circular** no domínio espacial. Como consequência, regiões opostas da imagem (topo e base, esquerda e direita) passam a interagir artificialmente, conforme ilustrado na Figura 5.8.
A aplicação de *zero-padding* antes da FFT reduz esse efeito ao estender a imagem com valores nulos nas bordas, aproximando o resultado da convolução linear. Esse comportamento pode ser interpretado à luz do Teorema da Convolução, apresentado na Figura 5.9.

```
%%writefile tmp/fig_05_padding_error.cpp
#define MM_OUT "tmp/fig_05_padding_error.png"
//| label: fig-05-padding-error
//| fig-cap: "Sem *padding*, um deslocamento severo faz a imagem vazar para o lado oposto
(convolução circular)."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
#include <complex>
#include <cmath>
#include <vector>
#include <string>
#include "morph.hpp"
#include <filesystem>

// Helper para converter mm::Image para cv::Mat
cv::Mat mmToMat(const mm::Image& img) {
    int type = (img.channels == 1) ? CV_8UC1 : CV_8UC3;
    return cv::Mat(img.h, img.w, type, const_cast<unsigned char*>(img.data.data()));
}

// Helper para fftshift manual
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));
    cv::Mat q1(mat, cv::Rect(cx, 0, cx, cy));
    cv::Mat q2(mat, cv::Rect(0, cy, cx, cy));
    cv::Mat q3(mat, cv::Rect(cx, cy, cx, cy));
    cv::Mat tmp;
    q0.copyTo(tmp); q3.copyTo(q0); tmp.copyTo(q3);
    q1.copyTo(tmp); q2.copyTo(q1); tmp.copyTo(q2);
}

// Helper para ifftshift manual
void ifftShift(cv::Mat& mat) {
    fftShift(mat); // ifftshift é o mesmo que fftshift para tamanhos pares/ímpares (inverso)
}

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 já está inicializado (fornecido externamente)

    //   Definir M e N
    int M = img_gray.h;
    int N = img_gray.w;

    // Simulação de um filtro de deslocamento brutal
    cv::Mat img_gray_float;
    mmToMat(img_gray).convertTo(img_gray_float, CV_32F);

    // Criar o filtro H_shift no domínio da frequência
    cv::Mat H_shift_real(M, N, CV_32F), H_shift_imag(M, N, CV_32F);
    for (int u = 0; u < M; u++) {
        for (int v = 0; v < N; v++) {
            double angle = -2.0 * M_PI * (u * 120.0 / M + v * 120.0 / N);
            H_shift_real.at<float>(u, v) = static_cast<float>(cos(angle));
            H_shift_imag.at<float>(u, v) = static_cast<float>(sin(angle));
        }
    }

    // Combinar partes real e imaginária em um único canal complexo
    cv::Mat planes_shift[] = {H_shift_real, H_shift_imag};
    cv::Mat H_shift;
    cv::merge(planes_shift, 2, H_shift);

    // Filtragem SEM padding (causa o wrap-around)
    cv::Mat F_img;
    cv::dft(img_gray_float, F_img, cv::DFT_COMPLEX_OUTPUT);

    // Multiplicar no domínio da frequência
    cv::Mat freq_planes[2];
    cv::split(F_img, freq_planes);
    cv::Mat shift_planes[2];
    cv::split(H_shift, shift_planes);

    cv::Mat result_real, result_imag;
    // (a+bi)*(c+di) = (ac-bd) + (ad+bc)i
    cv::multiply(freq_planes[0], shift_planes[0], result_real); // ac
    cv::Mat temp1;
    cv::multiply(freq_planes[1], shift_planes[1], temp1); // bd
    cv::subtract(result_real, temp1, result_real);
```

```
    cv::multiply(freq_planes[0], shift_planes[1], result_imag); // ad
    cv::Mat temp2;
    cv::multiply(freq_planes[1], shift_planes[0], temp2); // bc
    cv::add(result_imag, temp2, result_imag);

    cv::Mat result_planes[] = {result_real, result_imag};
    cv::Mat filtered_freq;
    cv::merge(result_planes, 2, filtered_freq);

    // Transformada inversa
    cv::Mat img_vazada_float;
    cv::idft(filtered_freq, img_vazada_float, cv::DFT_SCALE | cv::DFT_REAL_OUTPUT);

    // Normalizar para visualização
    cv::Mat img_vazada_vis;
    cv::normalize(img_vazada_float, img_vazada_vis, 0, 255, cv::NORM_MINMAX, CV_8U);

    // Converter de volta para mm::Image para exibição
    mm::Image output_img(img_vazada_vis.rows, img_vazada_vis.cols, 1);
    std::memcpy(output_img.data.data(), img_vazada_vis.data, output_img.data.size());

    // Exibir o resultado
    mm::show({img_gray, output_img}, MM_OUT,
             {"Original", "Filtragem s/ Padding (Vazamento)"}, 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] Filtragem s/ Padding (Vazamento)
```

```
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)")
```

Figura 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

#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]
```

```
    // Mesma suavização por dois caminhos: espaço vs. frequência.
    int M = img_gray.h;
    int N = img_gray.w;
    cv::Mat H = mm::gaussFilter(M, N, 30);      // H(u,v): transferência passa-baixa gaussiana
    mm::Image f_freq = mm::freqFilter(img_gray, H);    // convolução via multiplicação em
frequência
    cv::Mat img_gray_mat = img_gray;
    cv::Mat f_esp_mat;
    cv::GaussianBlur(img_gray_mat, f_esp_mat, cv::Size(31, 31), 5.0);
    mm::Image f_esp = f_esp_mat;

    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)")
```

Figura 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 Sobre a diferença numérica

A diferença residual da ordem de 10^{-13} não viola o Teorema da Convolução, mas reflete **limitações computacionais** inerentes à aritmética de ponto flutuante (precisão dupla, $\sim 10^{-16}$) e à **ordem de operações** entre os dois métodos:

- **Convolução espacial:** soma ponderada de vizinhos com arredondamentos sucessivos.
- **Convolução em frequência:** envolve três transformadas FFT e uma multiplicação complexa, sujeita a erros de truncamento e quantização.

Portanto, a igualdade teórica é exata, mas a implementação numérica produz uma diferença praticamente nula (erro relativo $< 10^{-12}$), confirmando o teorema dentro da precisão da máquina.

5.5 Filtros no Domínio da Frequência

Um filtro no domínio da frequência pode ser interpretado como uma **função de transferência aplicada ao espectro da imagem**. Nessa representação, cada coeficiente de frequência é multiplicado por um valor entre 0 e 1, que determina sua atenuação ou preservação. A forma dessa função define o efeito visual do filtro.

Corte abrupto e *ringing*. Filtros ideais com transição instantânea em uma frequência de corte D_0 produzem descontinuidades no domínio da frequência. Essa descontinuidade se reflete no domínio espacial como oscilações próximas a bordas, conhecidas como *ringing*. Esse efeito está associado à convolução com funções de suporte infinito no espaço, como a função *sinc*, conforme ilustrado na Figura 5.10.

Filtros com transição suave. Alternativas como os filtros Gaussiano e Butterworth suavizam a transição entre regiões de passagem e rejeição, reduzindo o *ringing*. Em contrapartida, essa suavização implica uma fronteira de separação menos definida entre frequências preservadas e atenuadas.

```
%%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 "morph.hpp"
#include <vector>
#include <string>
#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,
        std::vector<std::string>{
            "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)")
```

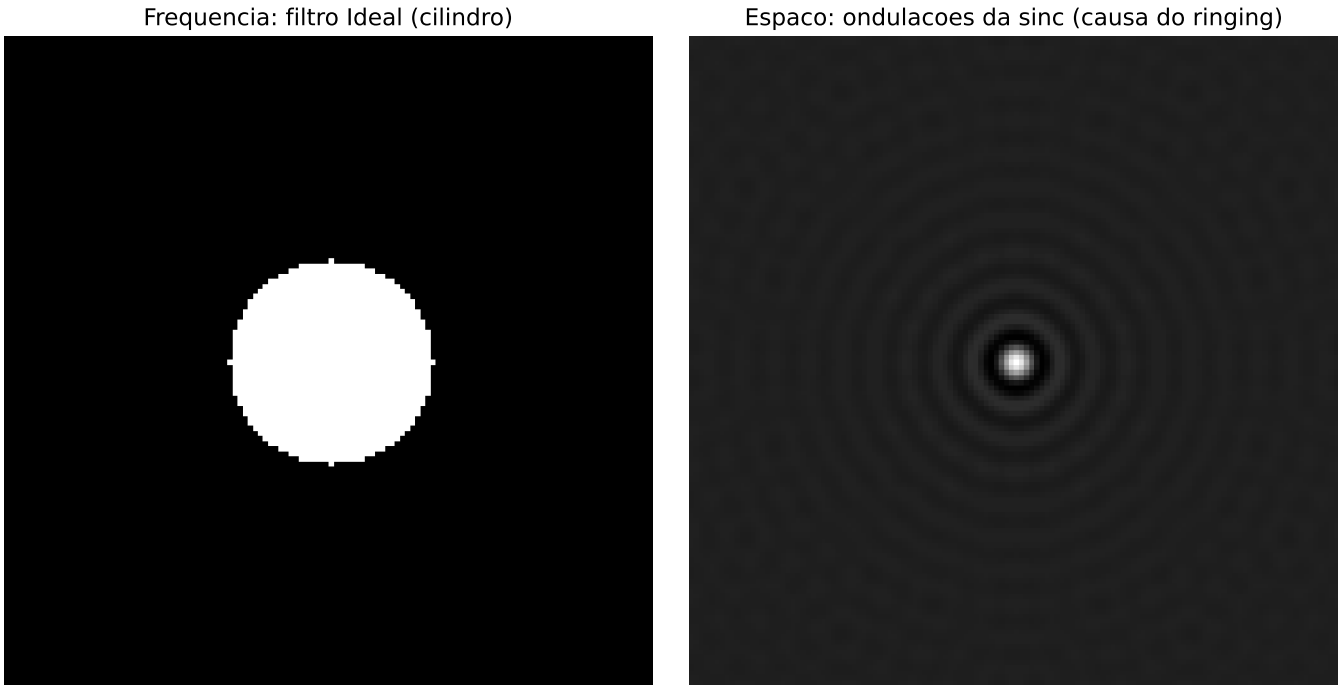


Figura 5.10: A Dualidade Perigosa: o corte abrupto na Frequência (cilindro Ideal) vira obrigatoriamente uma *sinc* no espaço. Suas ondulações causam o *ringing* fantasma nas bordas da imagem.

5.5.1 Filtros Passa-Baixa

Filtros passa-baixa atenuam componentes de alta frequência, resultando em suavização da imagem e redução de ruído. Após a centralização do espectro (FFT Shift), a distância de cada ponto ao centro é dada por:

$$D(u,v) = \sqrt{\left(u - \tfrac{M}{2}\right)^2 + \left(v - \tfrac{N}{2}\right)^2} \tag{5.5}$$

Filtro Ideal (LPFI):

$$H_{\text{ideal}}(u,v) = \begin{cases} 1, & D(u,v) \leq D_0 \\ 0, & D(u,v) > D_0 \end{cases} \tag{5.6}$$

O corte abrupto em D_0 introduz descontinuidades no domínio da frequência, resultando em oscilações no

domínio espacial conhecidas como *ringing*. Esse efeito está associado à convolução com funções de suporte infinito.

Filtro Gaussiano (LPFG):

$$H_{\text{gauss}}(u,v) = e^{-D^2(u,v)/(2\sigma^2)} \tag{5.7}$$

A suavidade da função Gaussiana no domínio da frequência evita descontinuidades, o que elimina o *ringing* e produz uma transição gradual entre frequências preservadas e atenuadas.

Filtro Butterworth (LPFB) de ordem n:

$$H_{\text{BW}}(u,v) = \frac{1}{1 + [D(u,v)/D_0]^{2n}} \tag{5.8}$$

O parâmetro n controla a suavidade da transição entre passagem e rejeição de frequências. Valores pequenos produzem transições suaves, enquanto valores grandes aproximam o comportamento do filtro ideal, com maior risco de *ringing*. Um exemplo comparativo é apresentado na Figura 5.11.

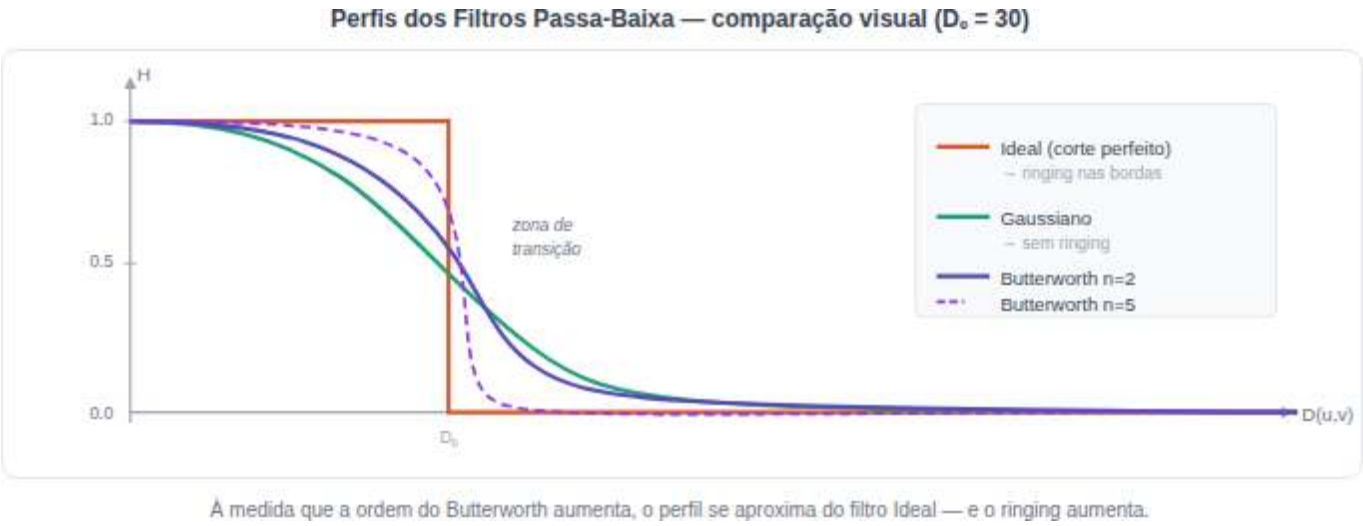


Figura 5.11: Filtros passa-baixa.

5.5.2 Filtros Passa-Alta e Passa-Banda

Filtros passa-alta podem ser obtidos a partir de um filtro passa-baixa complementar, definido como:

$$H_{\text{HP}}(u,v) = 1 - H_{\text{LP}}(u,v)$$

Esse tipo de filtro preserva componentes de alta frequência, realçando bordas e detalhes, enquanto atenua regiões de variação suave.

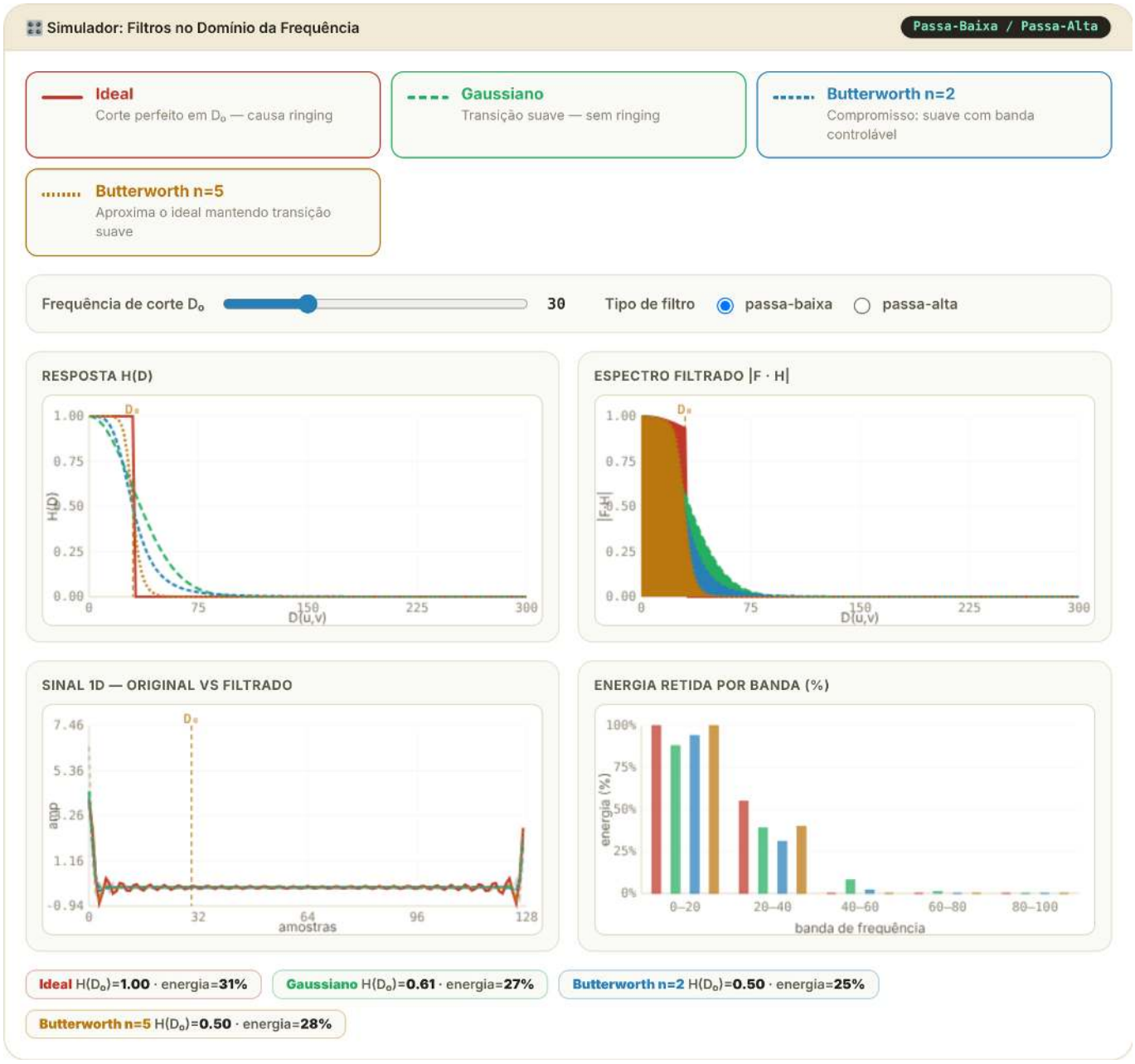
Filtros passa-banda preservam apenas uma faixa intermediária de frequências, limitada por dois raios D_L e D_H:

$$H_{\text{BP}}(u,v) = H_{\text{LP}}^{(D_H)}(u,v) \cdot \left[1 - H_{\text{LP}}^{(D_L)}(u,v)\right]$$

Esse tipo de filtragem é útil quando se deseja remover simultaneamente componentes de baixa e alta frequência, preservando apenas estruturas de escala intermediária.

Uma aplicação importante é a remoção de **ruído periódico**, no qual padrões regulares aparecem como picos localizados no espectro de magnitude. Esses picos podem ser atenuados por meio de filtros *notch* (rejeita-banda), posicionados especificamente nas frequências indesejadas.

Exemplos de filtros no domínio da frequência são apresentados no simulador da Figura 5.12, Figura 5.13 e Figura 5.14.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap05/sim-05-filtros.html

Figura 5.12: Simulador interativo de filtros no domínio da frequência.

```
%%writefile tmp/fig_05_filtros_freq.cpp
#define MM_OUT "tmp/fig_05_filtros_freq.png"
// g++ -std=c++17 -O2 snippet.cpp -o snippet $(pkg-config --cflags --libs opencv4)
#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <cmath>
#include "morph.hpp"

//| 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

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 by the cross-cell persistence system

    int M = img_gray.h;
    int N = img_gray.w;
    double D0 = 30;
    int n_bw = 2;

    //   Funções de transferência (H centrado) via construtores de morph.hpp
    //   Ideal:       1 se D<=D0, senão 0
    //   Gaussiano:   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);

    //   Imagens filtradas 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);

    // Função para visualizar H como imagem
    auto H_vis = [](const cv::Mat& H) -> mm::Image {
        cv::Mat H_norm;
        cv::normalize(H, H_norm, 0, 255, cv::NORM_MINMAX, CV_8U);
        return mm::Image(H_norm.rows, H_norm.cols, 1);
    };

    //   Perfis de H(u,v) na linha central, desenhados com cv2.line
    int linha = M / 2;
    int Wp = 512, Hp = 288;
    cv::Mat perfil = cv::Mat(Hp, Wp, CV_8UC3, cv::Scalar(255, 255, 255));

    auto traca = [&](const cv::Mat& row, cv::Scalar cor) {
        cv::Point ant;
        bool has_ant = false;
        for (int x = 0; x < N; x++) {
            int px = static_cast<int>(std::round(x * (Wp - 1) / (N - 1)));
            double val = row.at<double>(0, x);
            int py = static_cast<int>(std::round(10 + (1.0 - val) * (Hp - 20)));
            if (has_ant) {
                cv::line(perfil, ant, cv::Point(px, py), cor, 2, cv::LINE_AA);
            }
            ant = cv::Point(px, py);
            has_ant = true;
        }
    };

    cv::Mat row_ideal = H_ideal.row(linha);
    cv::Mat row_gauss = H_gauss.row(linha);
    cv::Mat row_bw = H_bw.row(linha);

    traca(row_ideal, cv::Scalar(216, 90, 48));   // (48, 90, 216) em BGR
    traca(row_gauss, cv::Scalar(29, 158, 117));  // (117, 158, 29) em BGR
```

```
    traca(row_bw, cv::Scalar(83, 74, 183));      // (183, 74, 83) em BGR

    // Converte perfil para mm::Image
    mm::Image perfil_img = mm::Image(perfil.rows, perfil.cols, perfil.channels());
    std::memcpy(perfil_img.data.data(), perfil.data, perfil_img.data.size());

    // Cria imagens de visualização para H
    cv::Mat H_ideal_vis, H_gauss_vis, H_bw_vis;
    cv::normalize(H_ideal, H_ideal_vis, 0, 255, cv::NORM_MINMAX, CV_8U);
    cv::normalize(H_gauss, H_gauss_vis, 0, 255, cv::NORM_MINMAX, CV_8U);
    cv::normalize(H_bw, H_bw_vis, 0, 255, cv::NORM_MINMAX, CV_8U);

    mm::Image H_ideal_img = mm::Image(H_ideal_vis.rows, H_ideal_vis.cols, 1);
    std::memcpy(H_ideal_img.data.data(), H_ideal_vis.data, H_ideal_img.data.size());

    mm::Image H_gauss_img = mm::Image(H_gauss_vis.rows, H_gauss_vis.cols, 1);
    std::memcpy(H_gauss_img.data.data(), H_gauss_vis.data, H_gauss_img.data.size());

    mm::Image H_bw_img = mm::Image(H_bw_vis.rows, H_bw_vis.cols, 1);
    std::memcpy(H_bw_img.data.data(), H_bw_vis.data, H_bw_img.data.size());

    std::vector<mm::Image> imagens = {
        img_gray, img_ideal, img_gauss, img_bw,
        H_ideal_img, H_gauss_img, H_bw_img, perfil_img
    };

    std::vector<std::string> titulos = {
        "Original", "LPF Ideal", "LPF Gaussiano", "LPF Butterworth (n=2)",
        "H Ideal", "H Gaussiano", "H Butterworth", "Perfis H(u,v)"
    };

    mm::show(imagens, MM_OUT, titulos, 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)")
```

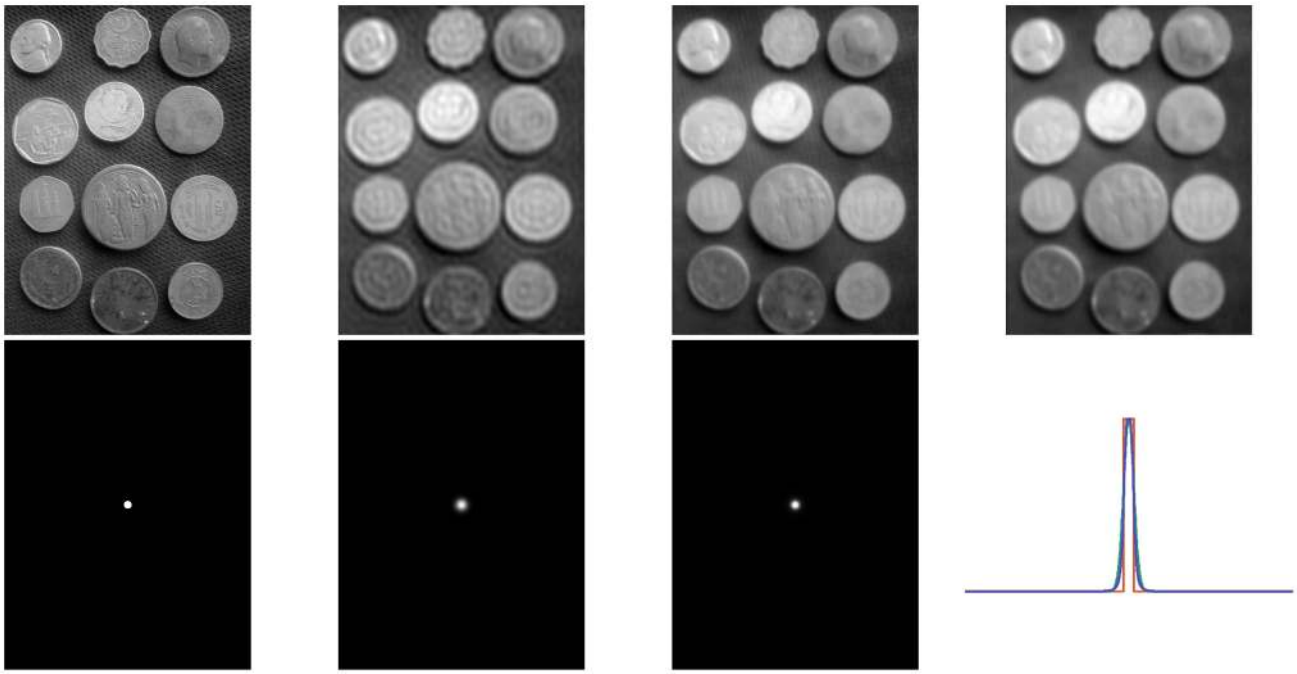


Figura 5.13: Comparação entre filtros passa-baixa: Ideal (D =30), Gaussiano (D =30) e Butterworth (D =30, n=2). Perfis de H(u,v) ao longo de uma linha central e imagens filtradas correspondentes.

```
%%writefile tmp/fig_05_filtros_passa_alta.cpp
#define MM_OUT "tmp/fig_05_filtros_passa_alta.png"
//| label: fig-05-filtros-passa-alta
//| fig-cap: "Filtro passa-alta Gaussiano. (a) Original; (b) Filtro passa-alta (D =30) - as
bordas das moedas e fundo texturizado são realçados."

#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]

    // Filtro passa-alta = complemento do passa-baixa Gaussiano (1 - H_gauss),
    // construído com mm.gaussFilter(..., highpass=true) e aplicado 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,
        {"Original", "Passa-alta Gaussiano ($D_0=30$)"},
        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] Passa-alta Gaussiano ($D_0=30$)
```

```
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)")
```

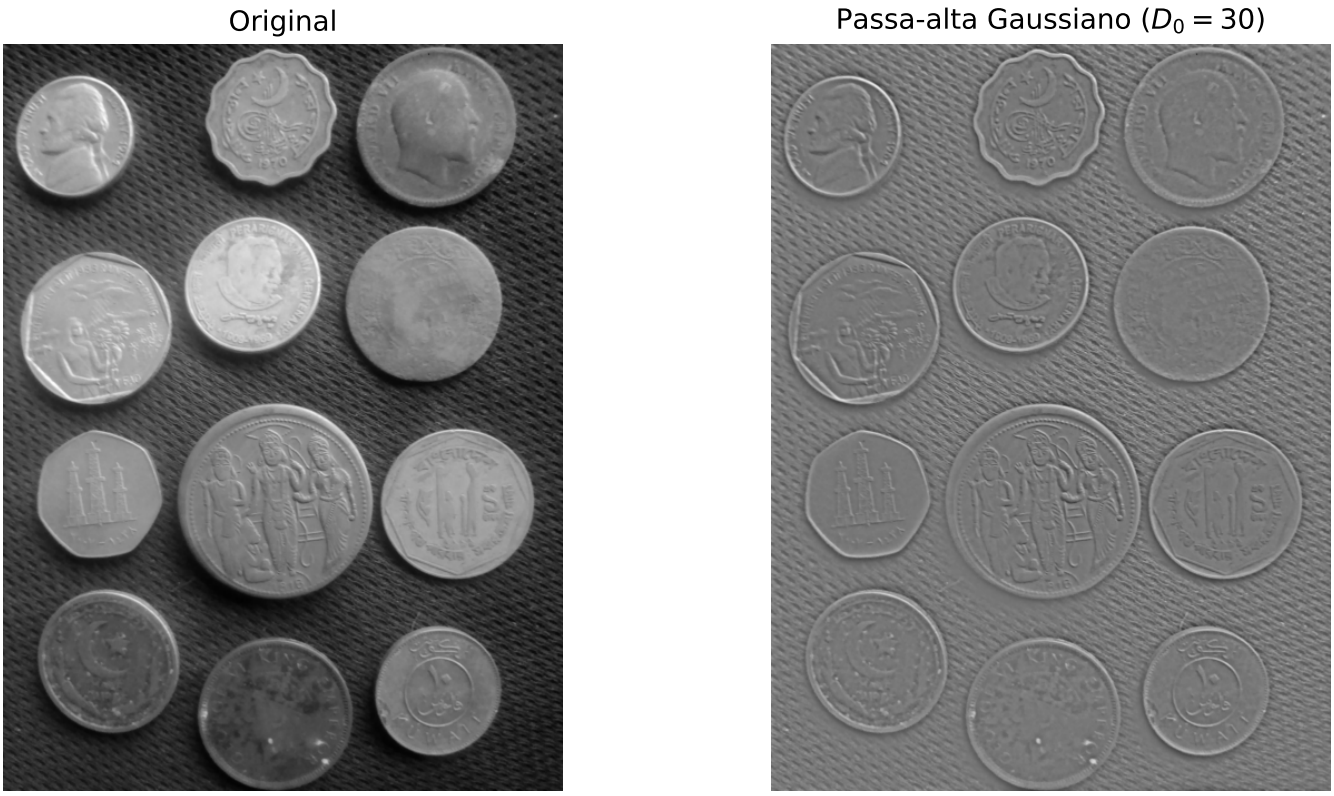


Figura 5.14: Filtro passa-alta Gaussiano. (a) Original; (b) Filtro passa-alta (D =30) - as bordas das moedas e fundo texturizado são realçados.

5.5.3 Remoção de Ruído Periódico

Ruído periódico — associado a interferências elétricas, padrões regulares de sensores ou artefatos de digitalização — aparece no espectro de Fourier como **picos pontuais simétricos em torno do centro**.

O filtro **rejeita-banda (*notch*)** atenua seletivamente essas frequências, preservando as demais componentes da imagem. Um exemplo de aplicação é apresentado na Figura 5.15.

```
%%writefile tmp/fig_05_ruido_periodico.cpp
#define MM_OUT "tmp/fig_05_ruido_periodico.png"
// Compile: g++ -std=c++17 -O2 $(pkg-config --cflags --libs opencv4) -o snippet snippet.cpp
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc.hpp>
#include <cmath>
#include <vector>
#include <string>
#include <cstdio>
#include "morph.hpp"

//| 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

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;

// Criar gradientes X e Y (coordenadas de malha)
cv::Mat X(h_img, w_img, CV_32F);
cv::Mat Y(h_img, w_img, CV_32F);
for (int j = 0; j < w_img; j++) {
    for (int i = 0; i < h_img; i++) {
        X.at<float>(i, j) = j;
        Y.at<float>(i, j) = i;
    }
}

int u0 = 20, v0 = 20;  // frequências exatas do ruído

// Construir ruído senoidal 2D
cv::Mat ruido(h_img, w_img, CV_64F);
for (int i = 0; i < h_img; i++) {
    for (int j = 0; j < w_img; j++) {
        double arg = 2.0 * M_PI * (u0 * j / (double)w_img + v0 * i / (double)h_img);
        ruido.at<double>(i, j) = 40.0 * std::sin(arg);
    }
}

// Converter imagem para float64 e adicionar ruído
cv::Mat img_gray_d = cv::Mat(h_img, w_img, CV_64F);
cv::Mat img_gray_8uc1 = cv::Mat(h_img, w_img, CV_8UC1, img_gray.data.data());
img_gray_8uc1.convertTo(img_gray_d, CV_64F);

cv::Mat img_ruidosa_d = img_gray_d + ruido;
cv::Mat img_ruidosa_8u;
cv::normalize(img_ruidosa_d, img_ruidosa_8u, 0, 255, cv::NORM_MINMAX, CV_8U);
mm::Image img_ruidosa(h_img, w_img, 1);
std::memcpy(img_ruidosa.data.data(), img_ruidosa_8u.data, img_ruidosa.data.size());

// 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 = cv::Mat::ones(h_img, w_img, CV_64F);
int r_notch = 8;
int cy = h_img / 2, cx = w_img / 2;

// Coordenadas de malha
cv::Mat yy(h_img, w_img, CV_64F);
cv::Mat xx(h_img, w_img, CV_64F);
for (int i = 0; i < h_img; i++) {
    for (int j = 0; j < w_img; j++) {
        yy.at<double>(i, j) = i;
        xx.at<double>(i, j) = j;
    }
}

// Pares de picos simétricos
std::vector<std::pair<int, int>> picos = {{v0, u0}, {-v0, -u0}, {v0, -u0}, {-v0, u0}};
for (const auto& pico : picos) {
    int dy = pico.first;
    int dx = pico.second;
    int center_y = cy + dy;
```

```
        int center_x = cx + dx;
        for (int i = 0; i < h_img; i++) {
            for (int j = 0; j < w_img; j++) {
                double dist_sq = (yy.at<double>(i, j) - center_y) * (yy.at<double>(i, j) -
center_y) +
                                 (xx.at<double>(i, j) - center_x) * (xx.at<double>(i, j) -
center_x);
                double dist = std::sqrt(dist_sq);
                if (dist <= r_notch) {
                    mascara.at<double>(i, j) = 0.0;
                }
            }
        }
    }
    cv::Mat mascara_8u;
    mascara.convertTo(mascara_8u, CV_8U, 255.0);
    mm::Image mascara_vis(h_img, w_img, 1);
    std::memcpy(mascara_vis.data.data(), mascara_8u.data, mascara_vis.data.size());

    // Filtragem: aplica a máscara centrada como filtro no domínio da frequência
    mm::Image img_rest_vis = mm::freqFilter(img_ruidosa, mascara);

    double psnr = cv::PSNR(img_gray_8uc1, (cv::Mat)img_rest_vis);
    char buf[256];
    std::snprintf(buf, sizeof(buf), "PSNR (original vs restaurada): %.2f dB", psnr);
    std::string psnr_str(buf);

    // Exibir imagens
    std::vector<mm::Image> imagens = {img_ruidosa, mag_vis, mascara_vis, img_rest_vis};
    std::vector<std::string> titles = {
        "Com ruído periódico",
        "Espectro (log)",
        "Máscara notch",
        "Restaurada (PSNR=" + [psnr] { char b[32]; std::snprintf(b, 32, "%.1f dB", psnr);
        return std::string(b); }() + ")"
    };
    mm::show(imagens, 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"
```

```
[1] Com ruído periódico
[2] Espectro (log)
[3] Máscara notch
[4] Restaurada (PSNR=33.2 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)")
```

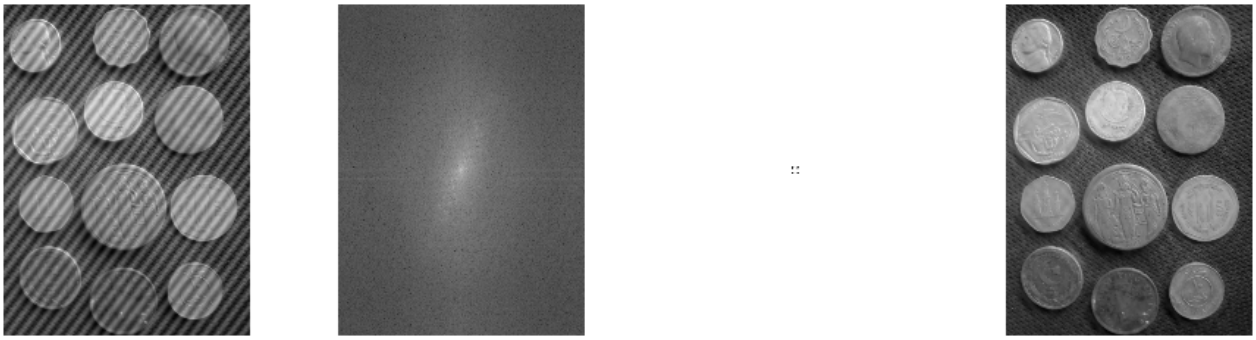


Figura 5.15: Remoção de ruído periódico via filtro *notch* no domínio da frequência: (a) imagem com ruído senoidal, (b) espectro mostrando os picos do ruído, (c) máscara *notch* centrada nos picos, (d) imagem restaurada.

📌 Síntese — Filtros Espectrais

Filtro	Efeito visual	Artefato	Uso
Passa-baixa ideal	Suavização intensa	*Ringing*	Ilustrativo
Passa-baixa Gaussiano	Suavização suave	Não apresenta *ringing*	Suavização geral
Passa-baixa Butterworth	Suavização controlada	*Ringing* (ordens altas)	Compromisso entre suavização e seletividade
Passa-alta	Realce de bordas	Amplificação de ruído	Detecção de contornos
Notch	Remoção seletiva de frequências	Possíveis distorções locais	Remoção de ruído periódico

O projeto de filtros no domínio da frequência consiste na definição de máscaras espectrais. Entretanto, efeitos no domínio espacial, como *ringing* e borramento, emergem diretamente dessas escolhas no espectro.

5.6 *Wavelets* e Multirresolução

A Transformada de Fourier decompõe o sinal em frequências **globais**: cada coeficiente $F(u, v)$ recebe contribuições de toda a imagem, sem informação explícita sobre a localização espacial dessas frequências. Assim, estruturas localizadas, como bordas, são representadas de forma distribuída no espectro.

As ***wavelets*** **(ondaletas)** superam essa limitação ao utilizar funções base **localizadas no espaço**, que podem ser deslocadas e escaladas. Essas funções possuem **suporte compacto**, isto é, são diferentes de zero apenas em uma região finita do domínio, permitindo uma representação simultânea em termos de **frequência e localização espacial**.

5.6.1 O Limite da Transformada de Fourier: localização espacial

A Transformada de Fourier descreve com precisão **quais frequências estão presentes** em um sinal, mas não representa explicitamente **onde essas frequências ocorrem no espaço**.

No experimento apresentado na Figura 5.16, duas imagens com estruturas localizadas em posições diferentes produzem espectros de magnitude praticamente idênticos. Isso ocorre porque a representação de Fourier é global: cada coeficiente recebe contribuição de toda a imagem.

Como consequência, o espectro de magnitude não representa explicitamente a localização de bordas ou outras estruturas, apenas a distribuição das frequências presentes. Essa limitação motivou o desenvolvimento de representações multirresolução, como a Transformada *Wavelet* Discreta (DWT), capazes de descrever simultaneamente a frequência e a localização espacial das estruturas da imagem.

```
%%writefile tmp/fig_05_fracasso_fourier.cpp
#define MM_OUT "tmp/fig_05_fracasso_fourier.png"
#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <cmath>
#include "morph.hpp"
#include <filesystem>

//| 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

int main() {
    // Cria os sinais de teste
    cv::Mat img_sinal1 = cv::Mat::zeros(128, 128, CV_32F);
    img_sinal1(cv::Rect(0, 20, 128, 5)).setTo(1.0f);
    img_sinal1(cv::Rect(100, 0, 5, 128)).setTo(1.0f);

    cv::Mat img_sinal2 = cv::Mat::zeros(128, 128, CV_32F);
    img_sinal2(cv::Rect(0, 90, 128, 5)).setTo(1.0f);
    img_sinal2(cv::Rect(30, 0, 5, 128)).setTo(1.0f);

    // Calcula os espectros de Fourier (transformada, shift e magnitude logarítmica)
    auto compute_log_magnitude = [](cv::Mat& src) -> cv::Mat {
        // FFT 2D
        cv::Mat padded;
        int m = cv::getOptimalDFTSize(src.rows);
        int n = cv::getOptimalDFTSize(src.cols);
        cv::copyMakeBorder(src, padded, 0, m - src.rows, 0, n - src.cols, cv::BORDER_CONSTANT,
cv::Scalar::all(0));

        cv::Mat planes[] = {padded, cv::Mat::zeros(padded.size(), CV_32F)};
        cv::Mat complexI;
        cv::merge(planes, 2, complexI);
        cv::dft(complexI, complexI);

        // Shift para o centro
        int cx = complexI.cols / 2;
        int cy = complexI.rows / 2;
        cv::Mat q0(complexI, cv::Rect(0, 0, cx, cy));
        cv::Mat q1(complexI, cv::Rect(cx, 0, complexI.cols - cx, cy));
        cv::Mat q2(complexI, cv::Rect(0, cy, cx, complexI.rows - cy));
        cv::Mat q3(complexI, cv::Rect(cx, cy, complexI.cols - cx, complexI.rows - cy));
        cv::Mat tmp;
```

```
        q0.copyTo(tmp);
        q3.copyTo(q0);
        tmp.copyTo(q3);
        q1.copyTo(tmp);
        q2.copyTo(q1);
        tmp.copyTo(q2);

        // Magnitude e log
        cv::Mat planes_shift[2];
        cv::split(complexI, planes_shift);
        cv::Mat mag;
        cv::magnitude(planes_shift[0], planes_shift[1], mag);
        mag += cv::Scalar::all(1.0f);
        cv::log(mag, mag);
        return mag;
    };

    cv::Mat mag1 = compute_log_magnitude(img_sinal1);
    cv::Mat mag2 = compute_log_magnitude(img_sinal2);

    // Normaliza para exibição
    cv::Mat img1_8u, img2_8u, mag1_8u, mag2_8u;
    cv::normalize(img_sinal1, img1_8u, 0, 255, cv::NORM_MINMAX, CV_8U);
    cv::normalize(img_sinal2, img2_8u, 0, 255, cv::NORM_MINMAX, CV_8U);
    cv::normalize(mag1, mag1_8u, 0, 255, cv::NORM_MINMAX, CV_8U);
    cv::normalize(mag2, mag2_8u, 0, 255, cv::NORM_MINMAX, CV_8U);

    // Converte para mm::Image e exibe
    std::vector<mm::Image> images;
    images.push_back(mm::Image(img1_8u.cols, img1_8u.rows, 1));
    std::memcpy(images[0].data.data(), img1_8u.data, images[0].data.size());

    images.push_back(mm::Image(mag1_8u.cols, mag1_8u.rows, 1));
    std::memcpy(images[1].data.data(), mag1_8u.data, images[1].data.size());

    images.push_back(mm::Image(img2_8u.cols, img2_8u.rows, 1));
    std::memcpy(images[2].data.data(), img2_8u.data, images[2].data.size());

    images.push_back(mm::Image(mag2_8u.cols, mag2_8u.rows, 1));
    std::memcpy(images[3].data.data(), mag2_8u.data, images[3].data.size());

    std::vector<std::string> titles = {"Sinal A", "Espectro A", "Sinal B (Deslocado)",
"Espectro B"};
    mm::show(images, 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)")
```

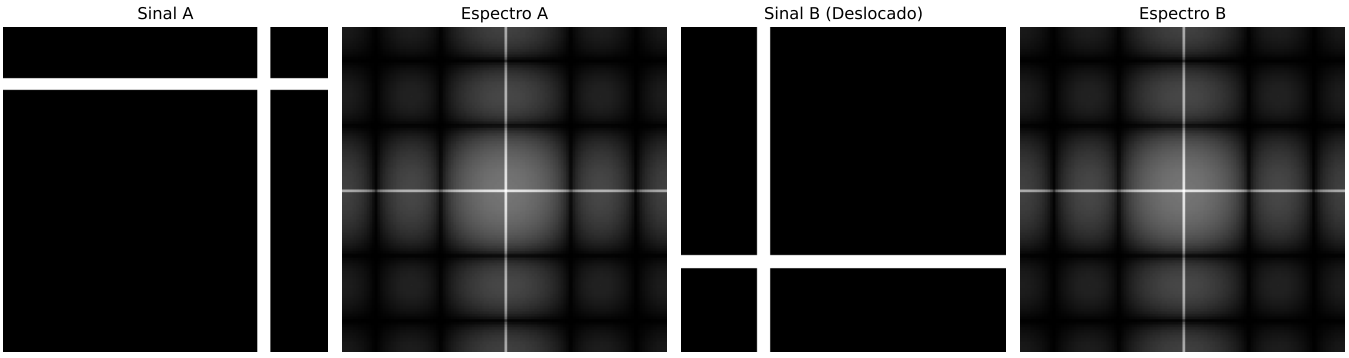


Figura 5.16: Fourier global é cego para a posição. Os espectros não dizem onde as bordas estão.

5.6.2 Transformada *Wavelet* Discreta 2D

A Transformada *Wavelet* Discreta (DWT) aplica, separadamente nas direções horizontal e vertical, dois filtros complementares: um **passa-baixa** h (aproximação) e um **passa-alta** g (detalhes), seguidos de subamostragem por um fator de 2 em cada dimensão. Esse processo produz quatro subbandas, cujos nomes indicam a combinação dos filtros aplicados em cada direção (L = *Low-pass*, passa-baixa; H = *High-pass*, passa-alta). As características de cada subbanda são resumidas na Tabela 5.3.

$$\mathrm{DWT}(f) = \{\ \underbrace{\mathrm{LL}}_{\text{aprox.}}\ ,\quad \underbrace{\mathrm{LH}}_{\text{detalhes horizontais}}\ ,\quad \underbrace{\mathrm{HL}}_{\text{detalhes verticais}}\ ,\quad \underbrace{\mathrm{HH}}_{\text{detalhes diagonais}}\ \}.$$

Tabela 5.3: Subbandas produzidas pela Transformada *Wavelet* Discreta 2D (DWT), indicando os filtros aplicados em cada direção e o conteúdo predominante de cada componente.

Subbanda	Filtros aplicados	Conteúdo visual
LL	baixa × baixa	Aproximação da imagem (versão suavizada e reduzida)
LH	baixa × alta	Bordas horizontais e variações verticais
HL	alta × baixa	Bordas verticais e variações horizontais
HH	alta × alta	Detalhes diagonais e texturas

A decomposição pode ser aplicada recursivamente sobre a subbanda LL, gerando uma representação multirresolução. Após J níveis, obtém-se uma estrutura com $3J + 1$ subbandas, em que cada novo nível reduz a resolução da componente de aproximação.

> **Conexão com CNNs**
>
> A decomposição multirresolução das *wavelets* possui uma relação conceitual com as representações hierárquicas utilizadas em redes neurais convolucionais (CNNs). Em ambos os casos, sucessivas etapas de filtragem e redução de resolução produzem descrições cada vez mais abstratas da imagem. Entretanto, as ***wavelets* utilizam filtros matematicamente definidos e reconstruíveis**, enquanto as **CNNs aprendem seus filtros durante o treinamento**.

5.6.3 Famílias de *Wavelets*

Diferentes famílias de *wavelets* apresentam compromissos distintos entre **suporte espacial**, suavidade e capacidade de compressão. O **suporte** corresponde à extensão da função *wavelet* no domínio espacial: quanto menor o suporte, mais localizada é a função; quanto maior, mais suave tende a ser sua representação, porém com maior custo computacional. A Tabela 5.4 compara algumas das famílias mais utilizadas.

Tabela 5.4: Comparação entre famílias de *wavelets*, destacando o comprimento do suporte, o número de momentos nulos, a simetria e aplicações típicas.

Wavelet	Comprimento do suporte	Momentos nulos	Simetria	Uso típico
Haar	2	1	Assimétrica	Introdução e análise básica
Daubechies db4	8	4	Assimétrica	Compressão e análise geral

Wavelet	Comprimento do suporte	Momentos nulos	Simetria	Uso típico
Symlet sym4	8	4	Quase simétrica	Reconstrução de sinais
Biortogonal 5/3	5/3	2/2	Simétrica	JPEG 2000 sem perda
Biortogonal 9/7	9/7	4/4	Simétrica	JPEG 2000 com perda

Os **momentos nulos** medem a capacidade da *wavelet* de representar regiões suaves da imagem com poucos coeficientes diferentes de zero. Uma *wavelet* com p momentos nulos anula exatamente polinômios de grau até $p-1$. Em consequência, quanto maior o número de momentos nulos, maior tende a ser a eficiência de compressão em regiões homogêneas, embora isso geralmente implique funções com suporte mais longo.

A Figura 5.17 apresenta as funções de base (*wavelets*) $\psi(t)$ no domínio espacial. Essas funções possuem **suporte compacto**, isto é, são diferentes de zero apenas em uma região finita do domínio, ao contrário das senoides da Transformada de Fourier, que se estendem por todo o domínio.

```
%%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 <vector>
#include <string>
#include "morph.hpp"
#include <filesystem>

int main() {
    // psi via mm.wavefun (algoritmo em cascata) - suporte compacto: decai a zero.
    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(x_h, std::vector<std::vector<double>>{psi_h}, {}, {(180,
60, 40)},
                                      "Ondaleta Haar (psi)", "t");
    mm::Image chart_d = mm::lineChart(x_d, std::vector<std::vector<double>>{psi_d}, {}, {(60,
140, 40)},
                                      "Ondaleta Daubechies 4 (psi)", "t");

    mm::show({chart_h, chart_d}, MM_OUT, {"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"
```

```
[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)")
```

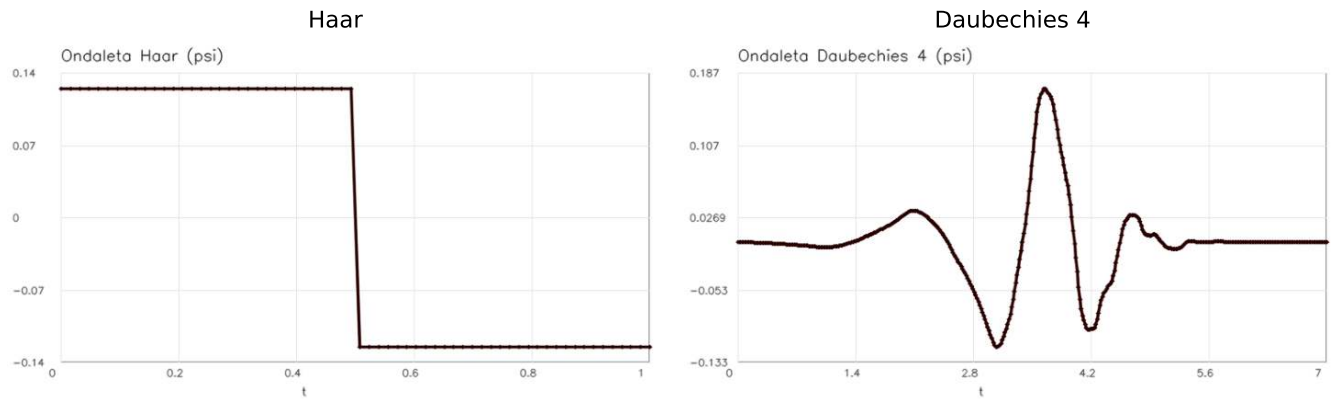


Figura 5.17: Funções da *Wavelet* ( ). Note como elas rapidamente decaem para zero (suporte compacto), ao contrário das senoides infinitas de Fourier.

O diagrama da Figura 5.18 ilustra a análise multirresolução realizada pela DWT, na qual a subbanda de aproximação (LL) é sucessivamente decomposta, formando uma representação hierárquica com dois níveis.

O simulador da Figura 5.19 permite explorar interativamente a Transformada *Wavelet* Discreta 2D (DWT) utilizando a *wavelet* de Haar. A decomposição em subbandas evidencia a separação entre a componente de aproximação e as componentes de detalhe da imagem.

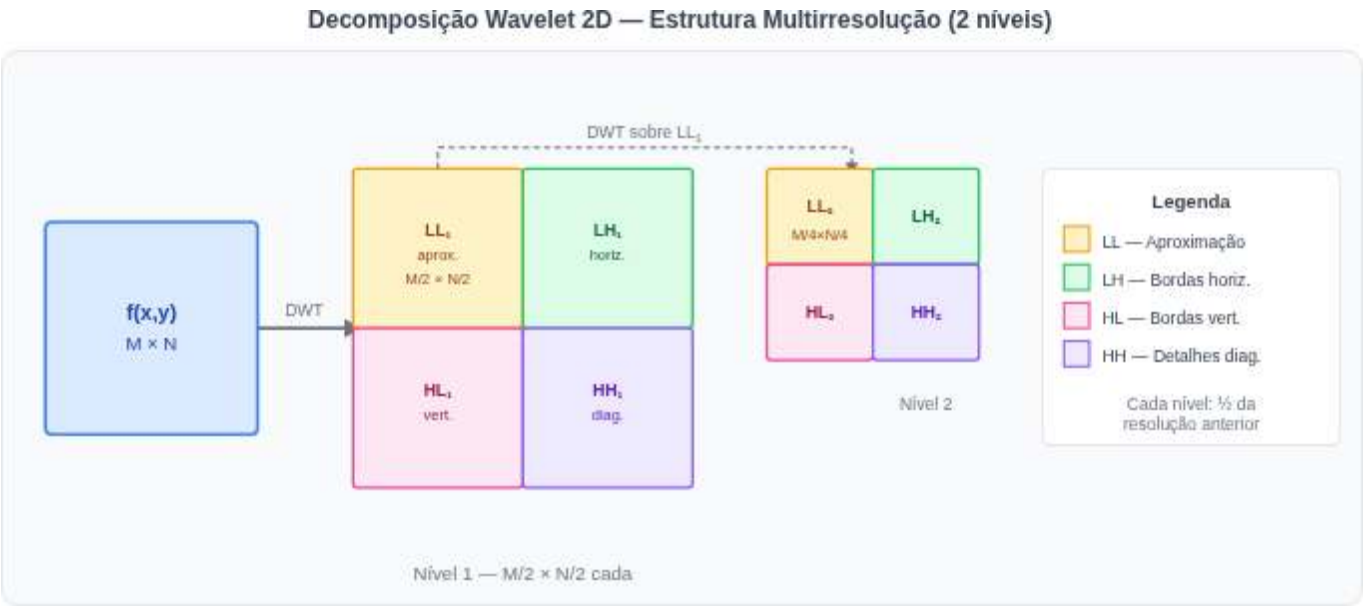


Figura 5.18: Diagrama da decomposição *wavelet* 2D em dois níveis.

Os diferentes padrões de entrada permitem observar o comportamento direcional dos filtros. Em imagens com bordas horizontais e verticais, as subbandas LH e HL destacam, respectivamente, as variações verticais e horizontais da intensidade. Em regiões de variação suave, a maior parte da energia concentra-se na subbanda de aproximação LL, enquanto as subbandas de detalhe apresentam coeficientes próximos de zero.

A análise multirresolução também pode ser observada ao aumentar o número de níveis de decomposição. Nesse caso, apenas a subbanda LL_1 é novamente decomposta, originando as subbandas LL_2, LH_2, HL_2 e HH_2, que formam o segundo nível da representação hierárquica.

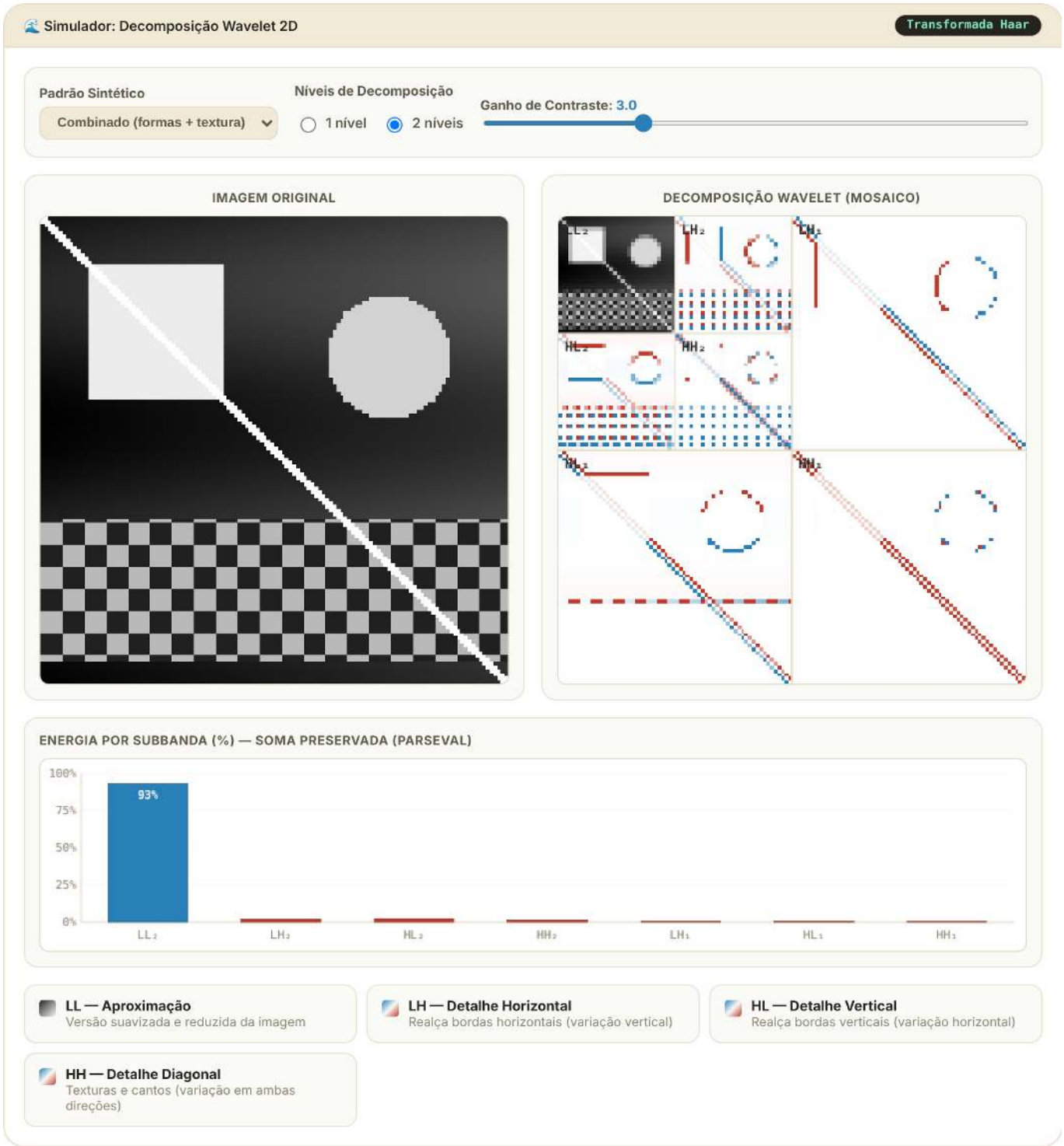
Em padrões formados por regiões homogêneas de grande extensão, como um degradê suave ou um tabuleiro composto por blocos grandes, a energia permanece predominantemente concentrada na subbanda LL. No degradê, isso ocorre porque as diferenças entre pixels vizinhos são pequenas. No tabuleiro, por sua vez, os pixels possuem praticamente a mesma intensidade no interior de cada bloco, de modo que apenas as fronteiras entre blocos produzem coeficientes não nulos nas subbandas de detalhe. Como essas fronteiras ocupam apenas uma pequena fração da imagem, sua contribuição para a energia total permanece reduzida.

Para viabilizar a análise visual dessas variações sutis, o simulador incorpora um controle de ganho de contraste dos detalhes (variando de 1 a 8). Esse parâmetro funciona como um fator de amplificação linear aplicado exclusivamente aos coeficientes das subbandas de detalhe (LH, HL e HH) antes de sua renderização em tela. Em cenários de transição suave (como o gradiente) ou de uniformidade local (como o interior dos blocos do tabuleiro), as diferenças numéricas calculadas pelo filtro passa-altas de Haar resultam em coeficientes muito próximos de zero, o que tornaria os quadrantes correspondentes escuros e imperceptíveis a olho nu. Ao multiplicar esses valores pelo ganho, o simulador resgata visualmente as estruturas de alta frequência ocultas e realça a orientação das bordas remanescentes.

O gráfico de energia por subbanda quantifica essa distribuição entre a componente de aproximação e as componentes de detalhe, demonstrando que o ganho visual não altera a métrica original da energia. Em imagens naturais, a maior parte da energia concentra-se na subbanda LL, enquanto as subbandas LH, HL e HH representam principalmente bordas, texturas e outras variações locais da intensidade.

5.6.4 Análise Multirresolução com a DWT 2D

A Transformada *Wavelet* Discreta 2D (DWT) decompõe uma imagem em componentes de aproximação e detalhe, organizadas de forma hierárquica em diferentes escalas e orientações. Como as subbandas de detalhe em imagens naturais frequentemente apresentam coeficientes de baixo contraste, os exemplos práticos a seguir utilizam um padrão geométrico sintético gerado em Python. Essa abordagem replica o comportamento do



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap05/sim-05-wavelet.html

Figura 5.19: Simulação da decomposição *wavelet* 2D.

simulador da Figura 5.19, tornando visualmente explícitos os efeitos da filtragem espacial e da decomposição multirresolução.

5.6.4.1 Decomposição em Mosaico de Múltiplos Níveis

A Figura 5.20 ilustra a estrutura hierárquica da DWT em dois níveis utilizando a *wavelet* de Haar. O processo baseia-se na aplicação combinada de filtros passa-baixa e passa-alta nas direções horizontal e vertical, seguidos por subamostragem por um fator de 2.

No primeiro nível, a imagem original origina a subbanda de aproximação (LL_1) e as componentes de detalhe horizontal (LH_1), vertical (HL_1) e diagonal (HH_1). Na análise multirresolução, a subbanda LL_1 é novamente filtrada e subamostrada, gerando o segundo nível de decomposição (LL_2, LH_2, HL_2 e HH_2).

Para viabilizar a interpretação visual das componentes de detalhe, o código extrai o valor absoluto de seus coeficientes e aplica uma normalização linear (*min-max*) para ocupar toda a faixa dinâmica de tons de cinza [0, 255]. Essa operação transforma regiões homogêneas (coeficientes nulos) em preto e destaca em branco as bordas e texturas extraídas em cada escala e orientação.

```
%%writefile tmp/fig_05_dwt_subbandas.cpp
#define MM_OUT "tmp/fig_05_dwt_subbandas.png"
//| 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

#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <cmath>
#include <iostream>
#include "morph.hpp"

//   Geração da Imagem Sintética (Mesmo padrão 'combined' do simulador)
cv::Mat gerar_imagem_sintetica(int N = 256) {
    cv::Mat img(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(y / 24.0);
            // 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;
            }
            v = std::max(0.0, std::min(255.0, v));
            img.at<double>(y, x) = v;
```

```
        }
    }
    cv::Mat img8;
    img.convertTo(img8, CV_8U);
    return img8;
}

// Normaliza subbanda para visualização [0,255]
cv::Mat sb_vis(const cv::Mat& sb) {
    cv::Mat sb_abs, sb_norm;
    cv::absdiff(sb, cv::Scalar(0), sb_abs);
    cv::normalize(sb_abs, sb_norm, 0, 255, cv::NORM_MINMAX);
    cv::Mat sb_u8;
    sb_norm.convertTo(sb_u8, CV_8U);
    return sb_u8;
}

int main() {
    //   Decomposição wavelet 2 níveis
    std::string wavelet = "haar";

    // Substitui a imagem escura de moedas pelo padrão sintético claro
    cv::Mat img_gray = gerar_imagem_sintetica(256);

    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 = {
        mm::Image(img_gray),
        mm::Image(sb_vis(LL1)), mm::Image(sb_vis(LH1)), mm::Image(sb_vis(HL1)),
mm::Image(sb_vis(HH1)),
        mm::Image(sb_vis(LL2)), mm::Image(sb_vis(LH2)), mm::Image(sb_vis(HL2)),
mm::Image(sb_vis(HH2))
    };

    std::vector<std::string> titles_dwt = {
        "Original",
        "LL1 (aprox.)", "LH1 (horiz.)", "HL1 (vert.)", "HH1 (diag.)",
        "LL2 (aprox.)", "LH2 (horiz.)", "HL2 (vert.)", "HH2 (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] LL1 (aprox.)
[3] LH1 (horiz.)
[4] HL1 (vert.)
[5] HH1 (diag.)
[6] LL2 (aprox.)
[7] LH2 (horiz.)
[8] HL2 (vert.)
[9] HH2 (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)")
```

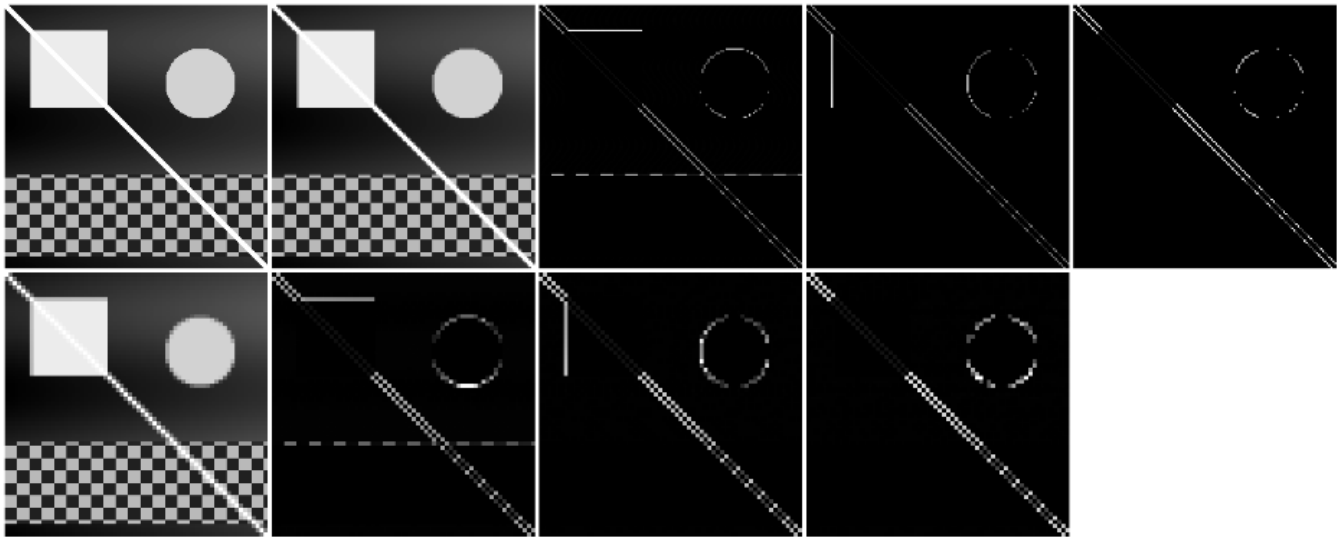


Figura 5.20: Decomposição *wavelet* 2D de 2 níveis com *wavelet* Haar: subbandas LL, LH, HL, HH em cada nível. As subbandas de detalhe revelam estruturas orientadas em diferentes escalas utilizando um padrão sintético.

5.6.4.2 O Compromisso entre Localização e Suavidade

A escolha da função de base (*wavelet*) influencia diretamente a forma como as feições da imagem são distribuídas e codificadas pelos coeficientes da DWT. A Figura 5.21 compara os resultados práticos obtidos ao aplicar quatro famílias distintas sobre o padrão geométrico sintético: `haar`, `db4`, `sym4` e `bior2.2`.

Por possuir suporte curto e formato de função degrau, a *wavelet* de Haar produz coeficientes altamente localizados nas descontinuidades espaciais, gerando bordas finas e nítidas nas subbandas de detalhe. Em contrapartida, famílias como Daubechies (`db4`) e Symlets (`sym4`), que apresentam maior suporte (filtros mais longos) e maior número de momentos nulos, geram respostas mais suaves e distribuídas ao redor das transições, o que pode introduzir leves oscilações ou borramentos nas fronteiras abruptas.

Esse comportamento evidencia o clássico compromisso (*trade-off*) da análise de multirresolução: suportes menores favorecem a localização espacial exata das bordas, enquanto suportes maiores e maior número de momentos nulos tendem a produzir representações mais esparsas e suaves. Essa suavidade e capacidade de atenuação de altas frequências garantem maior eficiência na compactação da energia, características fundamentais para aplicações de compressão de dados e remoção de ruído (*denoising*).

```
%%writefile tmp/fig_05_dwt_wavelets.cpp
#define MM_OUT "tmp/fig_05_dwt_wavelets.png"
#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <cmath>
#include <algorithm>
#include "morph.hpp"

//| label: fig-05-dwt-wavelets
//| fig-cap: "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."
//| echo: true
//| output: true

// Garante que img_gray e img_float utilizem o mesmo padrão sintético claro
```

```cpp
// Função auxiliar para visualizar subbandas (normalizar para 0-255)
static cv::Mat sb_vis(const cv::Mat& sb) {
    cv::Mat normalized;
    cv::normalize(sb, normalized, 0, 255, cv::NORM_MINMAX, CV_8U);
    return normalized;
}

// Função para gerar imagem sintética (fallback)
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 * ((double)x / N) + 15 * std::sin(y / 24.0);
            if (x > 24 && x < 100 && y > 24 && y < 100) v = 225;
            double 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;
            img.at<double>(y, x) = std::max(0.0, std::min(v, 255.0));
        }
    }
    cv::Mat img_u8;
    img.convertTo(img_u8, CV_8U);
    return img_u8;
}

int main() {
    // Gera imagem sintética
    cv::Mat img_gray = gerar_imagem_sintetica(256);

    cv::Mat img_float;
    img_gray.convertTo(img_float, CV_64F);

    std::vector<std::string> wavelets_comp = {"haar", "db4", "sym4", "bior2.2"};
    std::vector<mm::Image> imgs_comp;
    std::vector<std::string> titles_comp;

    for (const std::string& wname : wavelets_comp) {
        // Decomposição DWT 2D
        mm::Subbands s = mm::dwt2(img_float, wname);
        cv::Mat LL = s.LL;
        cv::Mat HH = s.HH;

        // Visualização das subbandas
        cv::Mat LL_vis = sb_vis(LL);
        cv::Mat HH_vis = sb_vis(HH);

        // Converter para mm::Image para exibição
        mm::Image img_ll(LL_vis.rows, LL_vis.cols, LL_vis.channels());
        std::memcpy(img_ll.data.data(), LL_vis.data, img_ll.data.size());
        mm::Image img_hh(HH_vis.rows, HH_vis.cols, HH_vis.channels());
        std::memcpy(img_hh.data.data(), HH_vis.data, img_hh.data.size());

        imgs_comp.push_back(img_ll);
        imgs_comp.push_back(img_hh);
        titles_comp.push_back(wname + " - LL ");
        titles_comp.push_back(wname + " - HH ");
    }
```

```
    // Exibir resultados
    std::vector<mm::Image> imgs_comp_display = imgs_comp;
    mm::show(imgs_comp_display, MM_OUT, titles_comp, 4);

    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"
```

```
[1] haar - LL
[2] haar - HH
[3] db4 - LL
[4] db4 - HH
[5] sym4 - LL
[6] sym4 - HH
[7] bior2.2 - LL
[8] bior2.2 - HH
```

```
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)")
```

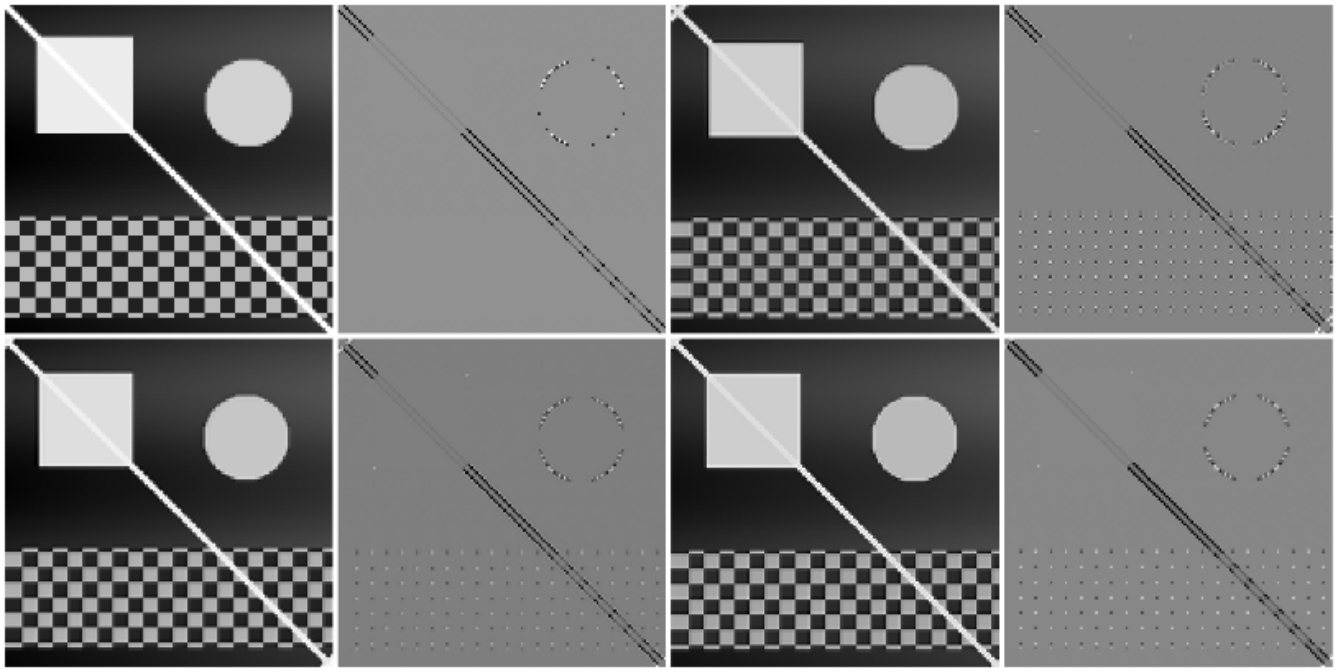


Figura 5.21: Comparação entre famílias de *wavelets*: Haar, db4, sym4 e bior2.2. Subbanda LL (aproximação) e HH (diagonal) para cada escolha, ilustrando o compromisso entre compactação e suavidade com base no padrão sintético.

5.6.4.3 Limiarização de Coeficientes e Compressão

Uma das principais aplicações da Transformada *Wavelet* Discreta (DWT) é a compressão de dados, impulsionada pela capacidade de representação **esparsas** dos coeficientes. A Figura 5.22 ilustra o efeito da limiarização abrupta (*hard thresholding*), técnica na qual coeficientes de detalhe com magnitude inferior a um limiar T são integralmente anulados antes do processo de síntese realizado pela Transformada *Wavelet* Discreta Inversa (IDWT).

À medida que o limiar T é elevado, um volume crescente de coeficientes de alta frequência é zerado. Por concentrarem menor energia, a remoção dessas componentes reduz consideravelmente a quantidade de informação necessária para representar a imagem, mantendo a componente de aproximação global (a subbanda LL mais profunda) intacta para preservar a estrutura macro. Visualmente, esse descarte de coeficientes manifesta-se através do desaparecimento progressivo de texturas finas e da suavização de transições abruptas de intensidade.

A fidelidade da imagem reconstruída frente à original é quantificada pela métrica de **Pico da Relação Sinal-Ruído (PSNR, *Peak Signal-to-Noise Ratio*)**, expressa em decibéis (dB). Valores mais altos de PSNR indicam menor distorção e maior proximidade matemática com o sinal original. O experimento prático evidencia o decaimento gradual do PSNR conforme a agressividade da limiarização aumenta, permitindo avaliar numericamente o limiar ótimo para o balanço entre compressão e degradação visual.

```
%%writefile tmp/fig_05_dwt_reconstrucao.cpp
#define MM_OUT "tmp/fig_05_dwt_reconstrucao.png"
#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <cmath>
#include <algorithm>
#include <iostream>
#include "morph.hpp"

//| 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

// Função para gerar a imagem sintética
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 * ((double)x / N) + 15 * std::sin(y / 24.0);
            if (24 < x && x < 100 && 24 < y && y < 100) v = 225;
            double 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;
            img.at<double>(y, x) = std::max(0.0, std::min(255.0, v));
        }
    }
    img.convertTo(img, CV_8U);
    return img;
}

// Função para aplicar hard thresholding em uma matriz
cv::Mat apply_hard_threshold(const cv::Mat& sb, double threshold) {
    cv::Mat result = sb.clone();
    for (int i = 0; i < result.rows; i++) {
        for (int j = 0; j < result.cols; j++) {
            double val = result.at<double>(i, j);
            if (std::abs(val) < threshold) {
                result.at<double>(i, j) = 0.0;
            }
        }
    }
    return result;
}

// Função para reconstruir após decomposição wavelet e thresholding
cv::Mat dwt_threshold_reconstruct(const cv::Mat& img, const std::string& wavelet = "db4", int
nivel = 2, double threshold = 0.0) {
    // Decompõe a imagem usando wavedec2
    cv::Mat img_float;
    img.convertTo(img_float, CV_64F);
    mm::WaveDec2 coefs = mm::wavedec2(img_float, wavelet, nivel);

    // Copia o LL final (não é limiarizado)
    std::vector<cv::Mat> coefs_t_ll;
    coefs_t_ll.push_back(coefs.LL);

    // Aplica hard thresholding a todos os detalhes
    std::vector<std::vector<cv::Mat>> coefs_t_details;
    for (int j = 0; j < nivel; j++) {
        std::vector<cv::Mat> detail_thr;
        detail_thr.push_back(apply_hard_threshold(coefs.detail[j][0], threshold)); // LH
        detail_thr.push_back(apply_hard_threshold(coefs.detail[j][1], threshold)); // HL
        detail_thr.push_back(apply_hard_threshold(coefs.detail[j][2], threshold)); // HH
        coefs_t_details.push_back(detail_thr);
```

```
    }

    // Reconstrução via waverec2
    mm::WaveDec2 coefs_t;
    coefs_t.LL = coefs_t_ll[0];
    coefs_t.detail = coefs_t_details;
    cv::Mat rec = mm::waverec2(coefs_t, wavelet);

    // Recorte para dimensão original
    rec = rec(cv::Rect(0, 0, img.cols, img.rows)).clone();

    // Clip e conversão para uint8
    cv::Mat rec_clipped;
    cv::Mat rec_float;
    rec.convertTo(rec_float, CV_64F);
    cv::max(0.0, rec_float, rec_float);
    cv::min(255.0, rec_float, rec_float);
    rec_float.convertTo(rec_clipped, CV_8U);
    return rec_clipped;
}

int main() {
    // Garante que img_gray utilize o mesmo padrão sintético claro
    cv::Mat img_gray;
    // Fallback caso o bloco anterior não tenha sido executado na mesma sessão
    img_gray = gerar_imagem_sintetica(256);

    std::vector<double> thresholds = {0, 10, 30, 60, 100};
    std::vector<cv::Mat> imgs_thr;
    imgs_thr.push_back(img_gray);
    std::vector<std::string> titles_thr;
    titles_thr.push_back("Original");

    for (double 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);

        // Formata o título com PSNR
        char buf[50];
        snprintf(buf, sizeof(buf), "T=%.0f  PSNR=%.1f dB", t, psnr);
        titles_thr.push_back(std::string(buf));
    }

    // Converte para mm::Image para exibição
    std::vector<mm::Image> imgs_out;
    for (const cv::Mat& m : imgs_thr) {
        mm::Image mm_img(m.rows, m.cols, 1);
        std::memcpy(mm_img.data.data(), m.data, mm_img.data.size());
        imgs_out.push_back(mm_img);
    }

    mm::show(imgs_out, 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)")
```

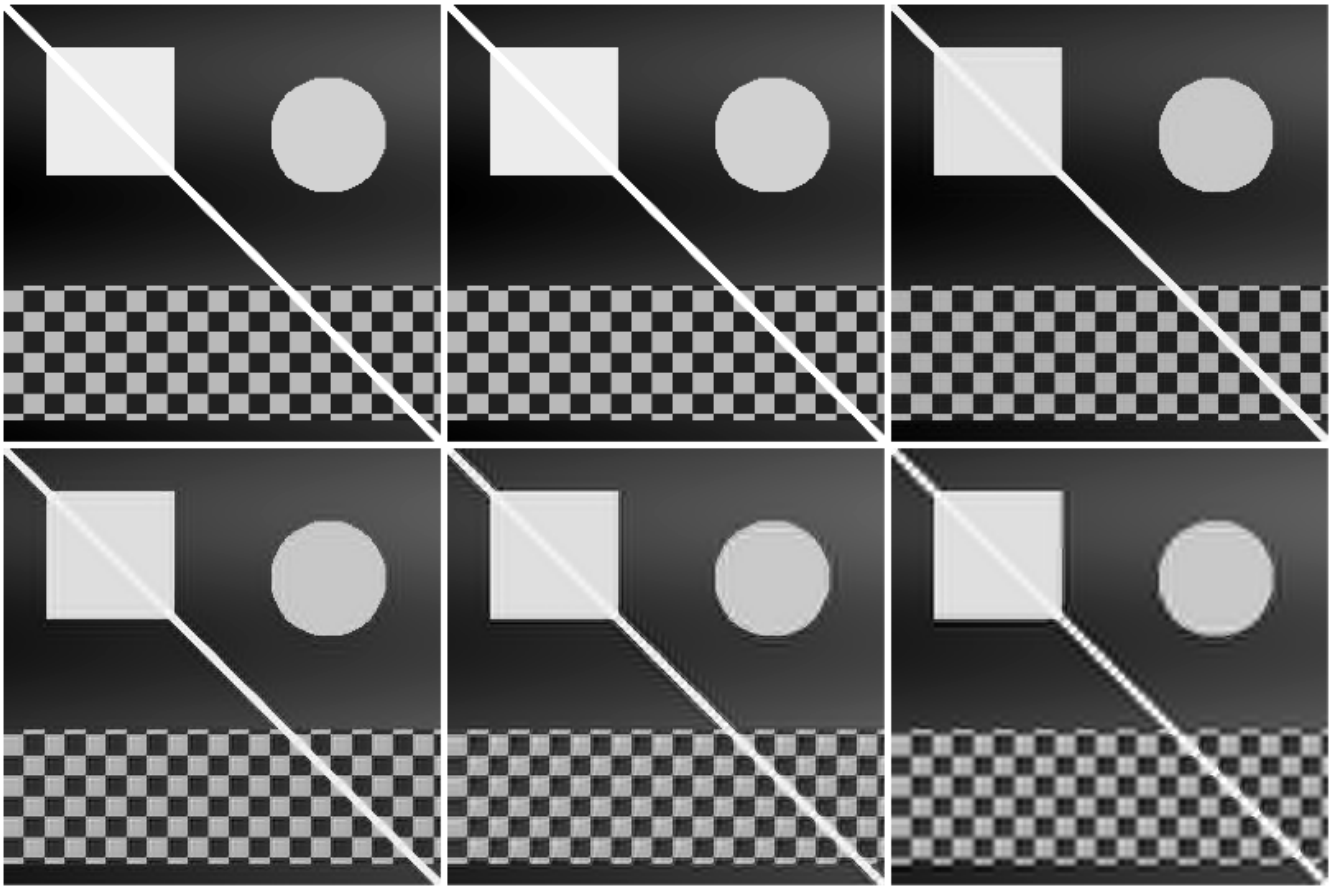


Figura 5.22: Reconstrução *wavelet* com limiarização de coeficientes (*hard thresholding*): à medida que o limiar aumenta, mais detalhes são zerados, produzindo imagens progressivamente mais suaves. Métrica PSNR quantifica a perda de qualidade sobre o padrão sintético.

Síntese — Fourier vs. *Wavelets*: quando utilizar cada abordagem?

A Tabela 5.5 sintetiza as principais diferenças estruturais e operacionais entre a Transformada Discreta de Fourier (DFT) e a Transformada *Wavelet* Discreta (DWT).

Tabela 5.5: Comparação entre a Transformada Discreta de Fourier (DFT) e a Transformada *Wavelet* Discreta (DWT), destacando suas principais características e aplicações.

Critério	Fourier (DFT)	*Wavelet* (DWT)
Funções de base	Senoides de suporte infinito	Funções de suporte compacto
Localização espacial	Não explícita (global)	Explícita (local)
Filtragem espectral	Excelente para controle fino de frequências	Baseada em subbandas (escalas)
Compressão de imagens	Base da DCT (JPEG tradicional)	Base da DWT (JPEG 2000)
Análise multiescala	Não	Sim
Remoção de ruído periódico	Altamente eficiente	Pouco indicada
Sinais não estacionários	Limitada	Altamente eficiente

Em termos práticos, a DFT consolida-se como a ferramenta ideal para análise espectral pura, projeto de filtros seletivos no domínio da frequência e atenuação de ruídos periódicos e harmônicos. Por outro lado, a DWT sobressai-se em cenários que exigem a preservação rigorosa da localização espacial das feições associada ao seu conteúdo frequencial, destacando-se em compressão de dados, análise multirresolução e processamento de transições abruptas. Desse modo, ambas as transformadas devem ser compreendidas como técnicas

perfeitamente complementares, mapeando caminhos distintos e específicos para a resolução de problemas em PDI-VC.

Analogias com Áudio: Limitações e Cuidados

Ao fazer analogias entre processamento de imagens e áudio, é importante considerar as diferenças fundamentais:

- Em sistemas de áudio estéreo/multicanais, a fase entre canais é crucial para a percepção de localização espacial (diferenças interaurais de fase e tempo).
- Em sistemas monaurais, a fase tem influência perceptual limitada — o ouvido humano é relativamente insensível à fase absoluta de componentes senoidais isolados.
- Em imagens, a fase da DFT é sempre fundamental para a localização espacial de estruturas, independentemente de ser uma imagem monocromática ou colorida.

A analogia entre fase em áudio e fase em imagens deve ser usada com cautela, destacando que, embora ambas carreguem informações sobre a organização espacial/temporal do sinal, os mecanismos perceptuais são fundamentalmente diferentes.

5.7 Compressão de Imagens

Enquanto as *wavelets* estabelecem a fundação teórica do padrão JPEG 2000, o padrão JPEG tradicional baseia-se na **Transformada Discreta de Cossenos (DCT, *Discrete Cosine Transform*)**. Apesar das diferenças estruturais, ambas as abordagens compartilham o mesmo princípio fundamental: compactar a energia da imagem em um número reduzido de coeficientes e descartar as componentes de menor relevância com impacto visual mínimo.

O objetivo central da compressão é reduzir o volume de dados necessário para o armazenamento ou transmissão de uma imagem. Esse processo é viabilizado pela identificação e eliminação de **redundâncias** estruturais e perceptuais.

5.7.1 Taxonomia das Redundâncias

O desenvolvimento de algoritmos de compressão fundamenta-se na identificação e eliminação de três categorias principais de redundância, sintetizadas na Tabela 5.6.

Tabela 5.6: Categorias de redundância em imagens digitais e seus respectivos mecanismos de exploração.

Tipo	Definição	Abordagem de Exploração
Espacial (interpixel)	Alta correlação e dependência estatística entre pixels vizinhos.	DCT, DWT e codificação preditiva.
Espectral (intercanal)	Correlação estatística entre os canais de cor de uma mesma imagem.	Transformações de espaço de cor (ex: RGB para YC_bC_r).
Psicovisual	Insensibilidade do sistema visual humano (SVH) a variações de alta frequência e baixo contraste.	Processos de quantização seletiva de coeficientes.

A depender da preservação da informação original após o processo de decodificação, os métodos de compressão dividem-se em duas classes fundamentais:

- **Sem perda (*lossless*):** Garante uma reconstrução bit a bit idêntica à imagem original. É empregada em cenários onde a integridade dos dados é estritamente crítica, como em imagens médicas, diagnósticos por imagem e armazenamento de documentos textuais.
- **Com perda (*lossy*):** Admite a introdução de uma distorção controlada no sinal em troca de taxas de compressão substancialmente mais elevadas. É a abordagem padrão para fotografias de consumo e

streaming de vídeo, ecossistemas nos quais o SVH tolera pequenas atenuações de alta frequência sem percepção de degradação da qualidade visual.

5.7.2 Transformada de Cossenos Discreta (DCT-II 2D)

A **Transformada de Cossenos Discreta** (DCT) constitui a operação central do padrão JPEG. Diferentemente da DFT, que utiliza uma base complexa, a DCT baseia-se em funções trigonométricas puramente reais. Para um bloco de imagem $f(x, y)$ de dimensões $N \times N$, a **DCT-II 2D** mapeia o sinal espacial para o domínio das frequências espaciais, gerando a matriz de coeficientes $C(u, v)$ por meio de:

$$C(u,v) = \alpha(u)\,\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y)\, \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right] \tag{5.9}$$

onde os fatores de normalização ortogonal são dados por $\alpha(0) = \sqrt{1/N}$ e $\alpha(k) = \sqrt{2/N}$ para $k > 0$.

Cada coeficiente $C(u, v)$ quantifica a contribuição — ou "peso" — de uma frequência espacial específica dentro daquele bloco. O termo $C(0, 0)$ é denominado **componente DC** e representa a intensidade média do bloco (frequência nula). Os demais coeficientes, chamados de **componentes AC** (*Alternating Current*), correspondem às frequências espaciais progressivamente maiores.

5.7.3 As Funções de Base da DCT

Sob uma perspectiva geométrica, a Equação 5.9 realiza a projeção do bloco de pixels sobre um conjunto de funções ortogonais. Para o caso padrão do JPEG ($N = 8$), o bloco espacial é decomposto em uma combinação linear de **64 funções de base** bidimensionais, denotadas por $B_{u,v}(x, y)$ e geradas pelo produto de funções cossenoidais:

$$B_{u,v}(x,y) = \cos\left[\frac{\pi(2x+1)u}{16}\right] \cos\left[\frac{\pi(2y+1)v}{16}\right]$$

Dessa forma, a operação inversa pode ser interpretada como a reconstrução exata do bloco original por meio da soma ponderada dessas 64 matrizes de base, onde cada coeficiente $C(u, v)$ atua como o peso analítico de sua respectiva componente harmônica.

A **frequência espacial** indicada pelos índices (u, v) determina o número de ciclos de oscilação ao longo das dimensões horizontais e verticais do bloco. Como ilustrado na Figura 5.23 — cujo código isola cada base aplicando a transformação inversa sobre impulsos unitários —, essas 64 funções são organizadas em uma matriz 8×8. O canto superior esquerdo ($u = 0, v = 0$) exibe o padrão uniforme de frequência nula (DC), enquanto o avanço para a direita (eixo u) ou para baixo (eixo v) mapeia variações harmônicas progressivamente maiores, representando transições rápidas, bordas e texturas nas orientações horizontais, verticais e diagonais.

i DCT vs DFT: Vantagem da Compactação de Energia

Tanto a DCT quanto a DFT mapeiam um bloco $N \times N$ espacial em uma matriz de coeficientes de mesma dimensão. Contudo, para imagens naturais, a DCT apresenta maior eficiência na **compactação de energia** nas baixas frequências. Isso ocorre porque a DCT assume implicitamente uma simetria par do sinal nas fronteiras do bloco, o que equivale a uma extensão periódica contínua, minimizando o efeito de espalhamento espectral (*ringing*). Como consequência, a maioria dos coeficientes AC decai rapidamente para valores próximos de zero, otimizando o *pipeline* de compressão sem introduzir degradação visual perceptível.

```
%%writefile tmp/fig_05_dct_basis.cpp
#define MM_OUT "tmp/fig_05_dct_basis.png"
#include <opencv2/opencv.hpp>
```

```cpp
#include <vector>
#include <string>
#include <cstring>
#include "morph.hpp"
#include <filesystem>

//| label: fig-05-dct-basis
//| fig-cap: "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."
//| echo: true
//| output: true

int main() {
    // Cada base é a IDCT de um único coeficiente unitário - montadas num mosaico 8×8.
    int tile = 32;
    cv::Mat montagem = cv::Mat::zeros(8 * tile, 8 * tile, CV_8U);

    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_8u;
            cv::normalize(base, base_8u, 0, 255, cv::NORM_MINMAX, CV_8U);

            cv::Mat base_resized;
            cv::resize(base_8u, base_resized, cv::Size(tile, tile), 0, 0, cv::INTER_NEAREST);

            // Copiar a base redimensionada para o mosaico
            cv::Mat roi = montagem(cv::Rect(j * tile, i * tile, tile, tile));
            base_resized.copyTo(roi);
        }
    }

    std::vector<mm::Image> images;
    mm::Image result(montagem.rows, montagem.cols, montagem.channels());
    std::memcpy(result.data.data(), montagem.data, result.data.size());
    images.push_back(result);

    std::vector<std::string> titles = {"As 64 bases da DCT-II 8×8 (DC no topo-esquerdo)"};
    mm::show(images, 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] As 64 bases da DCT-II 8×8 (DC no topo-esquerdo)
```

```
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)")
```

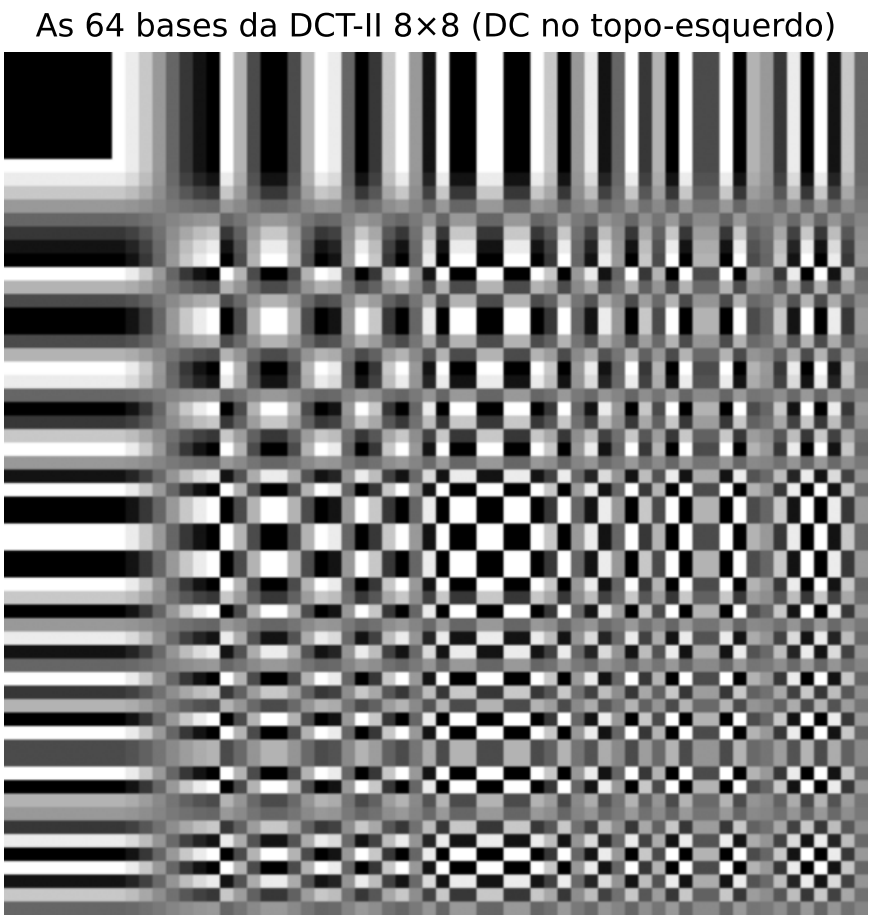


Figura 5.23: O Alfabeto Visual do JPEG: As 64 funções de base da DCT-II. O coeficiente DC fica no topo esquerdo (suave). Ao descer e avançar à direita, a oscilação espacial aumenta drasticamente.

5.7.4 Concentração de Energia e Reconstrução Progressiva

Antes da aplicação da DCT, os pixels do bloco de intensidade são rotineiramente transladados (subtraindo-se 128 para imagens de 8 bits) a fim de centralizar o sinal em torno de zero, eliminando componentes contínuas desnecessárias. Ao computar a DCT sobre o bloco resultante, a propriedade de **compactação de energia** torna-se evidente: a quase totalidade da variância e da informação da imagem original concentra-se no coeficiente DC ($C(0,0)$) e nos primeiros harmônicos AC de baixa frequência.

A Figura 5.24 demonstra esse fenômeno por meio de uma reconstrução progressiva por truncamento abrupto. Em vez de utilizar todos os 64 coeficientes, o algoritmo preserva apenas os k primeiros componentes — selecionados com base em uma varredura que prioriza as baixas frequências espaciais — e anula os demais.

A síntese inversa (**IDCT**) realizada com apenas uma fração dos coeficientes (como 15% ou 30%) já é capaz de recuperar as estruturas e a iluminação macro do bloco original de pixels. À medida que harmônicos de frequências mais altas são progressivamente reincorporados, os detalhes finos e as transições rápidas são restaurados. Esse comportamento valida o princípio da compressão perceptual: as altas frequências descartadas possuem pouca energia e sua ausência, em condições normais, gera um impacto visual secundário na percepção do observador.

```
%%writefile tmp/fig_05_dct_bloco.cpp
#define MM_OUT "tmp/fig_05_dct_bloco.png"
// Compile: g++ -std=c++17 -o program main.cpp $(pkg-config --cflags --libs opencv4)

#include <opencv2/opencv.hpp>
#include <iostream>
```

```
#include <vector>
#include <string>
#include <cmath>
#include <algorithm>
#include "morph.hpp"

//| label: fig-05-dct-bloco
//| fig-cap: "DCT 2D em bloco 8×8: coeficientes e reconstrução progressiva."
//| echo: true
//| output: true

// Função para DCT-II 2D ortogonal (separável)
cv::Mat dct2(const cv::Mat& bloco) {
    cv::Mat temp, result;
    cv::dct(bloco, temp);  // DCT nas linhas
    cv::dct(temp.t(), result);  // DCT nas colunas
    return result.t();
}

// Função para IDCT-II 2D ortogonal
cv::Mat idct2(const cv::Mat& coefs) {
    cv::Mat temp, result;
    cv::idct(coefs, temp);  // IDCT nas linhas
    cv::idct(temp.t(), result);  // IDCT nas colunas
    return result.t();
}

// Função para ordenar índices por soma (zig-zag simplificado)
std::vector<std::pair<int,int>> getZigZagIndices() {
    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) {
                  int sum_a = a.first + a.second;
                  int sum_b = b.first + b.second;
                  if (sum_a != sum_b) return sum_a < sum_b;
                  // Para mesma soma, prioriza diagonal secundária
                  return a.first > b.first;
              });
    return indices;
}

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 é fornecido automaticamente como mm::Image

    // Converter mm::Image para cv::Mat
    cv::Mat img_mat(img_gray.h, img_gray.w, CV_8UC1, img_gray.data.data());

    //   Bloco 8×8 centralizado da imagem
    int cy = img_gray.h / 2;
    int cx = img_gray.w / 2;

    cv::Mat bloco_raw = img_mat(cv::Rect(cx, cy, 8, 8)).clone();
```

```
    cv::Mat bloco_float;
    bloco_raw.convertTo(bloco_float, CV_32F);
    bloco_float -= 128.0f;  // Centralizar em torno de zero

    // Aplicar DCT 2D
    cv::Mat C = dct2(bloco_float);

    std::cout << "Coeficientes DCT do bloco 8×8:" << std::endl;
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            std::cout << (int)std::round(C.at<float>(i, j)) << " ";
        }
        std::cout << std::endl;
    }

    // Calcular energias
    double energia_dc = C.at<float>(0, 0) * C.at<float>(0, 0);
    double energia_total = 0;
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            energia_total += C.at<float>(i, j) * C.at<float>(i, j);
        }
    }

    std::cout << "\nEnergia DC     : " << energia_dc << std::endl;
    std::cout << "Energia total  : " << energia_total << std::endl;
    std::cout << "Fração no DC   : " << (energia_dc / energia_total) * 100 << "% ←
concentração de energia" << std::endl;

    //   Reconstrução progressiva
    std::vector<mm::Image> imgs_rec;
    std::vector<std::string> titles_rec;

    // Bloco original
    cv::Mat bloco_original;
    bloco_raw.convertTo(bloco_original, CV_8U);
    mm::Image img_orig(8, 8, 1);
    std::memcpy(img_orig.data.data(), bloco_original.data, 64);
    imgs_rec.push_back(img_orig);
    titles_rec.push_back("Bloco original\n(8×8 pixels)");

    // Obter índices em ordem zig-zag
    std::vector<std::pair<int,int>> indices = getZigZagIndices();

    // Testar diferentes números de coeficientes
    std::vector<int> keeps = {1, 4, 10, 20, 40, 64};
    for (int keep : keeps) {
        cv::Mat C_trunc = cv::Mat::zeros(8, 8, CV_32F);
        for (int k = 0; k < keep; k++) {
            int u = indices[k].first;
            int v = indices[k].second;
            C_trunc.at<float>(u, v) = C.at<float>(u, v);
        }

        // Reconstruir bloco
        cv::Mat rec_float = idct2(C_trunc);
        rec_float += 128.0f;

        // Clipping e conversão para uint8
        cv::Mat rec;
        rec_float.convertTo(rec, CV_8U);
```

```
        cv::threshold(rec, rec, 255, 255, cv::THRESH_TRUNC);

        mm::Image img_rec(8, 8, 1);
        std::memcpy(img_rec.data.data(), rec.data, 64);
        imgs_rec.push_back(img_rec);

        int pct = (int)std::round((keep / 64.0) * 100);
        titles_rec.push_back(std::to_string(keep) + " coef.\n(" + std::to_string(pct) + "% do
total)");
    }

    // Exibir resultados
    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:
12 17 17 16 15 14 11 7
19 26 23 20 19 17 14 8
22 29 25 22 21 20 16 8
22 31 32 35 38 35 24 11
21 31 37 47 53 46 29 13
21 31 35 44 52 47 31 13
19 27 28 35 45 45 34 18
12 16 16 22 29 33 28 16

Energia DC     : 143.593
Energia total  : 49504
Fração no DC   : 0.290063% ← 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.
(63% 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)")
```

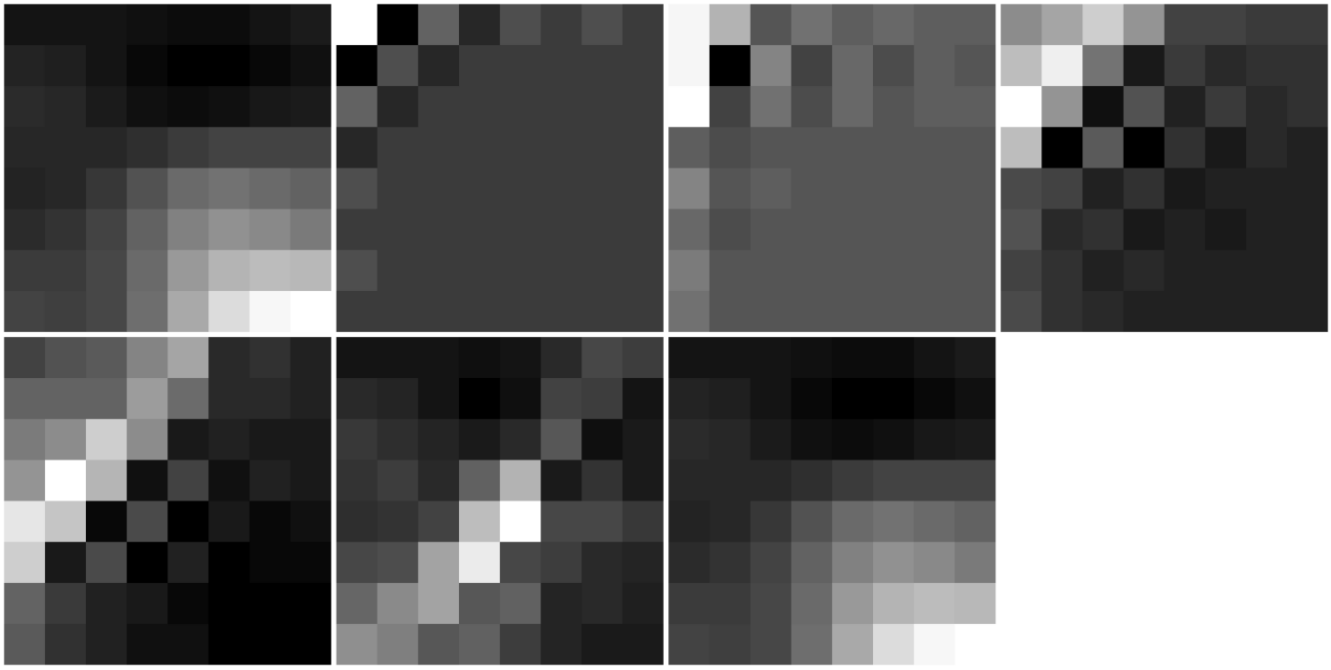


Figura 5.24: DCT 2D em bloco 8×8: coeficientes e reconstrução progressiva.

5.7.5 O *Pipeline* de Compressão JPEG

O padrão JPEG opera dividindo a imagem em blocos disjuntos de 8×8 pixels, processados por meio de uma sequência de transformações espaciais, perceptuais e estatísticas. O *pipeline* completo de codificação é estruturado em seis etapas principais:

$$\text{RGB} \xrightarrow{\text{(1) } YC_bC_r} \xrightarrow{\text{(2) Subamostragem}} \xrightarrow{\text{(3) Blocos 8×8}} \xrightarrow{\text{(4) DCT}} \xrightarrow{\text{(5) Quantização}} \xrightarrow{\text{(6) Codificação Entrópica}}$$

A Tabela 5.7 detalha a função analítica e o fundamento perceptual que justifica cada uma dessas etapas.

Tabela 5.7: Etapas do *pipeline* de compressão JPEG e seus respectivos fundamentos de projeto.

Etapa	Operação	Fundamento Perceptual e Estatístico
1	Conversão $RGB \to YC_bC_r$	Separa a luminância (Y) da crominância (C_b, C_r). O sistema visual humano (SVH) apresenta maior sensibilidade a variações de brilho do que de cor.

Etapa	Operação	Fundamento Perceptual e Estatístico
2	Subamostragem de crominância (ex: 4:2:0)	Reduz a resolução espacial dos canais de cor pela metade, descartando dados redundantes com impacto visual desprezível.
3–4	Centralização e aplicação da DCT 8×8	Translada os pixels para o intervalo $[-128, 127]$ e compacta a energia espectral do bloco nos coeficientes de baixa frequência.
5	Quantização linear seletiva	Divide cada coeficiente $C(u, v)$ pelo elemento correspondente da matriz $Q(u, v)$, aplicando arredondamento inteiro. Constitui a principal fonte de compressão com perda.
6	Varredura em ziguezague e codificação	Ordena os coeficientes quantizados para maximizar sequências nulas consecutivas, otimizando a codificação por comprimento de corrida (RLE) e a codificação de Huffman.

A **matriz de quantização** $Q(u, v)$ é o mecanismo central de controle do compromisso entre taxa de compressão e qualidade visual. No algoritmo prático da Figura 5.25, o fator de qualidade estipulado pelo usuário (escala de 1 a 100) é convertido em um escalar que parametriza a severidade da matriz Q. Valores reduzidos de qualidade expandem os divisores de $Q(u, v)$, forçando o truncamento em massa dos coeficientes AC para zero. Quando essa eliminação é excessiva, a descontinuidade nas fronteiras dos blocos adjacentes não é atenuada na reconstrução, gerando os denominados **artefatos de bloco** (*blocking artifacts*).

A Lógica da Varredura em Ziguezague

A eficiência do codificador entrópico subsequente à quantização depende diretamente da ordenação dos dados. Como a DCT concentra a energia vital no vértice superior esquerdo da matriz (baixas frequências) e empurra os coeficientes nulos para as extremidades opostas, a leitura linear por linhas ou colunas fragmentaria as sequências de zeros.

A ordenação em ziguezague soluciona essa limitação ao percorrer a matriz diagonalmente em ordem crescente de frequência espacial. Esse mapeamento agrupa os coeficientes significativos no início do vetor e concentra os coeficientes nulos em uma única sequência contínua ao final do arranjo, permitindo que o algoritmo RLE codifique grandes blocos de dados de forma compacta e eficiente.

> **i** O que é RLE?
>
> **RLE** (*Run-Length Encoding*) é uma técnica de compressão sem perdas que codifica sequências consecutivas de valores idênticos — especialmente **zeros** — como um par (contagem, valor). No JPEG, após a varredura em ziguezague, os coeficientes quantizados são organizados de modo que os zeros se concentrem ao final do vetor. O RLE então comprime essa longa corrida de zeros com extrema eficiência, otimizando o armazenamento e a transmissão da imagem comprimida.

```
%%writefile tmp/fig_05_jpeg_pipeline.cpp
#define MM_OUT "tmp/fig_05_jpeg_pipeline.png"
```

```
#include <opencv2/opencv.hpp>
#include <string>
#include <vector>
#include <cstdio>
#include "morph.hpp"

//| 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

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_resized;
    cv::Mat img_read = cv::imread("imagens/cameraman.png", cv::IMREAD_GRAYSCALE);
    cv::resize(img_read, img_resized, cv::Size(256, 256));
    mm::Image img_gray_mm = mm::gray(mm::Image(img_resized));
    mm::Image img_src = img_gray_mm;

    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);
        char title_buf[128];
        std::snprintf(title_buf, sizeof(title_buf), "Q=%d (PSNR=%.1f dB)", q,
mm::psnr(img_src, rec));
        titles.push_back(std::string(title_buf));
    }

    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=30.7 dB)
```

```
[4] Q=50 (PSNR=32.8 dB)
[5] Q=75 (PSNR=35.2 dB)
[6] Q=90 (PSNR=40.0 dB)
```

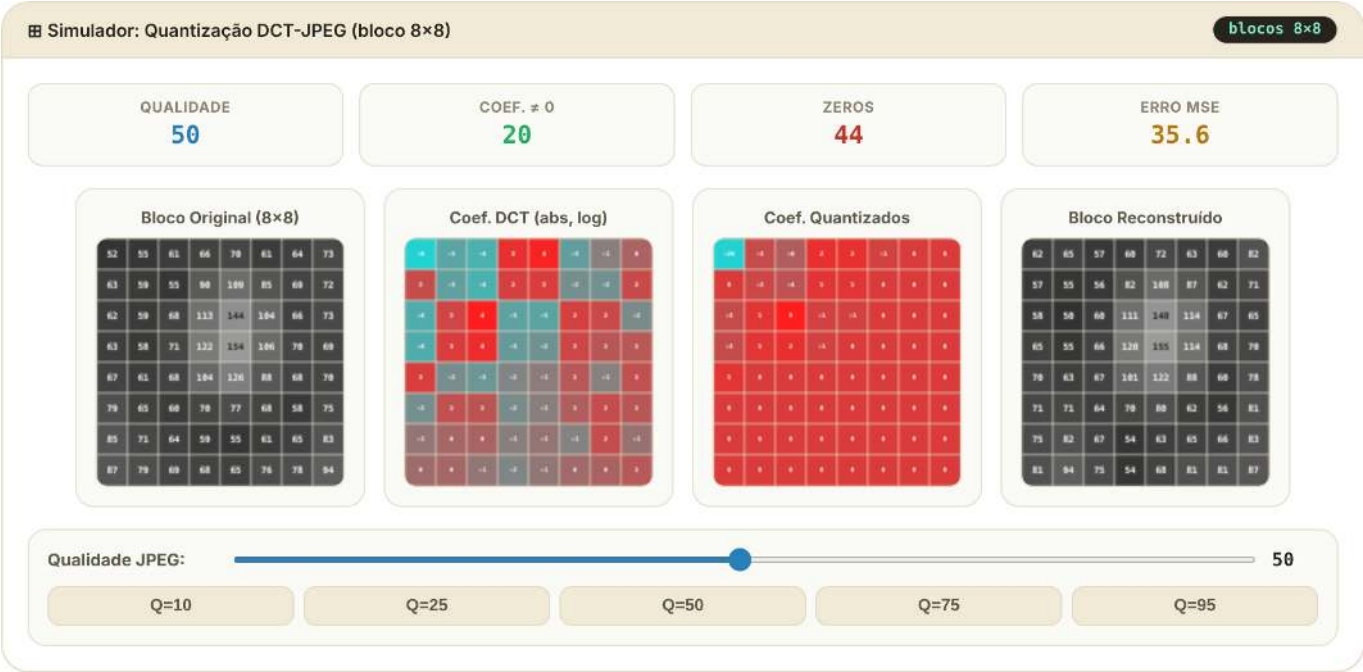
```
try:
    mm.show(mm.read("tmp/fig_05_jpeg_pipeline.png"), figsize=(14, 10))
except Exception as _e:
    print("figura indisponivel nesta trilha (C++): " + repr(_e) + "
    tmp/fig_05_jpeg_pipeline.png (ver a versao Python)")
```

Figura 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 Simulador Interativo: Quantização DCT

O simulador da Figura 5.26 permite explorar o impacto do processo de quantização sobre um bloco 8×8 extraído de uma imagem real, sintetizando em tempo real as seguintes componentes:

- **Bloco original e reconstruído:** Representação direta dos pixels no domínio espacial em escala de cinza [0, 255].
- **Coeficientes DCT:** Distribuição da energia mapeada de forma logarítmica em um gradiente cromático, evidenciando a concentração de intensidade no vértice superior esquerdo (baixas frequências).
- **Coeficientes quantizados:** Exibição dos valores inteiros resultantes da divisão pela matriz $Q(u, v)$, tornando visualmente explícito o surgimento em massa de coeficientes nulos (em tons escuros) conforme o fator de qualidade é reduzido.
- **Métricas de compressão:** Painel de monitoramento que quantifica o Erro Quadrático Médio (MSE), o número de coeficientes preservados e o volume de zeros gerados para a codificação entrópica.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap05/sim-05-dct.html

Figura 5.26: Simulador interativo de compressão DCT-JPEG: ajuste o fator de qualidade e visualize em tempo real os coeficientes zerados, o bloco reconstruído e o erro de quantização.

5.8 Comparação de Formatos de Imagem

A escolha de um formato de armazenamento digital impacta diretamente o compromisso entre qualidade visual, tamanho de arquivo e custo computacional de decodificação. Os três formatos de maior relevância para arquiteturas *web* e sistemas de computação visual são o JPEG, o PNG e o WebP.

5.8.1 Características dos Formatos

A Tabela 5.8 sintetiza as propriedades estruturais dos principais formatos de imagem rasterizados.

Tabela 5.8: Comparação estrutural entre os principais formatos de imagem rasterizados.

Característica	JPEG	PNG	WebP
Compressão	Com perda	Sem perda	Com e sem perda.
Transparência (canal alfa)	Não	Sim	Sim.
Suporte a animação	Não	Limitado (APNG)	Sim.
Algoritmo base	DCT + Huffman	DEFLATE (LZ77 + Huffman)	VP8 / VP8L.
Melhor para	Fotografia	Gráficos, texto e ícones	Uso universal em ambiente Web.
Pior para	Texto e bordas nítidas	Imagens fotográficas complexas	Compatibilidade legada.

5.8.2 Métricas de Avaliação de Qualidade

Duas métricas objetivas são amplamente adotadas para quantificar a distorção introduzida por processos de compressão:

Pico da Relação Sinal-Ruído (PSNR, *Peak Signal-to-Noise Ratio*):

$$\text{PSNR} = 10 \log_{10}\left(\frac{L^2}{\text{MSE}}\right) \quad [\text{dB}] \tag{5.10}$$

onde $L = 255$ para imagens quantizadas em 8 bits e MSE representa o **Erro Quadrático Médio** (*Mean Squared Error*). Valores de PSNR acima de 40 dB indicam excelente fidelidade; entre 30 dB e 40 dB representam boa qualidade; e valores inferiores a 30 dB correspondem a degradações visuais facilmente perceptíveis.

Índice de Similaridade Estrutural (SSIM, *Structural Similarity Index*):

$$\text{SSIM}(f, g) = \frac{(2\mu_f\mu_g + c_1)(2\sigma_{fg} + c_2)}{(\mu_f^2 + \mu_g^2 + c_1)(\sigma_f^2 + \sigma_g^2 + c_2)} \tag{5.11}$$

O SSIM avalia janelas locais da imagem com base em três componentes complementares: **luminância** (μ_f, μ_g), **contraste** (σ_f, σ_g) e **estrutura** (σ_{fg}), ponderados por constantes de estabilidade c_1 e c_2. O índice varia no intervalo $[-1, 1]$, onde a unidade representa a identidade perfeita. Ao contrário do PSNR, o SSIM considera a organização espacial dos erros, alinhando-se à percepção do sistema visual humano (SVH).

> **i** PSNR vs SSIM: Aplicação de Métricas Perceptuais
>
> O PSNR possui formulação matemática simples e baixo custo computacional, contudo, tende a superestimar a qualidade em imagens com distorções localizadas ou subestimá-la em variações globais de brilho toleradas pelo observador. O SSIM modela com maior fidelidade a percepção biológica, mas exige maior esforço de processamento. Para análises rigorosas de codificadores, recomenda-se reportar ambas as métricas estatísticas em caráter complementar.

5.8.3 Inspeção Visual: Natureza dos Artefatos de Compressão

A natureza matemática do codificador dita o tipo de degradação introduzida em taxas de bits reduzidas. Conforme ilustrado na Figura 5.27, a compressão agressiva via DCT no padrão JPEG segmenta a imagem em malhas rígidas, gerando os **artefatos de bloco** (*blocking artifacts*). Em contrapartida, algoritmos baseados em codificação preditiva ou representações submetidas a transformadas espaciais avançadas (como o WebP e o JPEG 2000) eliminam as descontinuidades de bloco, mas introduzem perda de textura fina e borramentos característicos ao redor de bordas de alto contraste.

```
%%writefile tmp/fig_05_zoom_artefatos.cpp
#define MM_OUT "tmp/fig_05_zoom_artefatos.png"
// Compile: g++ -std=c++17 -o main main.cpp `pkg-config --cflags --libs opencv4`
#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include "morph.hpp"
#include <filesystem>

#ifndef MM_OUT
#endif

//| 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
```

```cpp
int main() {
    // Carrega a imagem e converte para escala de cinza
    mm::Image img_loaded = mm::read("imagens/cameraman.png");
    cv::Mat img_src_cv = img_loaded;
    cv::Mat img_resized;
    cv::resize(img_src_cv, img_resized, cv::Size(256, 256));
    mm::Image img_src = mm::gray(cv::Mat(img_resized));

    // Salva com baixa qualidade JPEG e WebP
    std::vector<int> jpeg_params;
    jpeg_params.push_back(cv::IMWRITE_JPEG_QUALITY);
    jpeg_params.push_back(10);
    std::vector<int> webp_params;
    webp_params.push_back(cv::IMWRITE_WEBP_QUALITY);
    webp_params.push_back(10);

    cv::Mat img_src_mat = img_src;
    cv::imwrite("tmp/zoom_q10.jpg", img_src_mat, jpeg_params);
    cv::imwrite("tmp/zoom_q10.webp", img_src_mat, webp_params);

    // Função de zoom
    auto zoom = [](cv::Mat& img) {
        cv::Mat roi = img(cv::Range(120, 200), cv::Range(150, 230));
        cv::Mat resized;
        cv::resize(roi, resized, cv::Size(320, 320), 0, 0, cv::INTER_NEAREST);
        return resized;
    };

    // Lê as imagens comprimidas
    cv::Mat img_src_mat2 = img_src;
    cv::Mat jpeg_img = cv::imread("tmp/zoom_q10.jpg", cv::IMREAD_GRAYSCALE);
    cv::Mat webp_img = cv::imread("tmp/zoom_q10.webp", cv::IMREAD_GRAYSCALE);

    // Aplica zoom
    cv::Mat z_orig_cv = zoom(img_src_mat2);
    cv::Mat z_jpeg_cv = zoom(jpeg_img);
    cv::Mat z_webp_cv = zoom(webp_img);

    // Converte para mm::Image para exibição
    mm::Image z_orig = z_orig_cv;
    mm::Image z_jpeg = z_jpeg_cv;
    mm::Image z_webp = z_webp_cv;

    std::vector<mm::Image> images = {z_orig, z_jpeg, z_webp};
    std::vector<std::string> titles = {
        "Zoom Original",
        "JPEG Q=10 (Artefato de Bloco)",
        "WebP Q=10 (Suavizacao)"
    };
    mm::show(images, 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)")
```

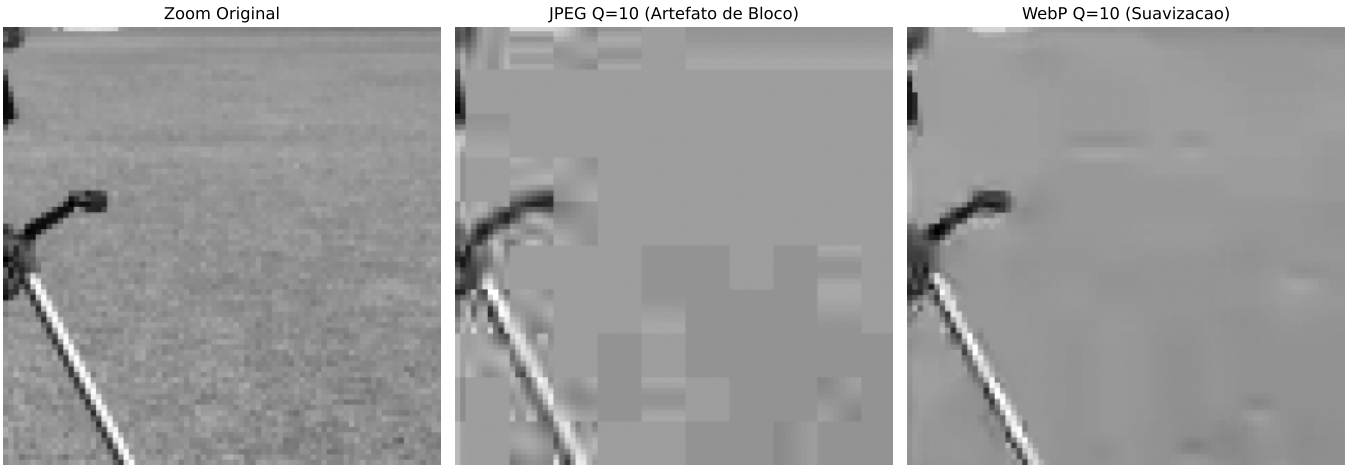


Figura 5.27: Análise comparativa de artefatos de compressão sob fator de qualidade reduzido ($Q = 10$). À esquerda, observa-se o artefato de bloco característico da discretização por DCT no JPEG. À direita, evidencia-se o efeito de atenuação e suavização de bordas intrínseco ao padrão WebP.

5.8.4 Avaliação Quantitativa e Espacial da Compressão

A validação dos algoritmos de compressão com perda exige uma análise que correlacione o custo de armazenamento à fidelidade do sinal reconstruído. Essa avaliação é realizada de forma complementar através de curvas de desempenho global e pelo mapeamento local das distorções induzidas pelos codificadores.

5.8.4.1 Curvas de Taxa-Distorção

A Figura 5.28 apresenta a avaliação empírica do *pipeline* JPEG e WebP por meio de **curvas de taxa-distorção**, que monitoram o ganho de compressão (tamanho do arquivo em KB) em função do PSNR. O formato PNG atua como linha de base ideal (PSNR $= \infty$), pois sua natureza *lossless* impede qualquer degradação, embora demande um volume de dados substancialmente maior.

A análise das curvas demonstra a superioridade e a eficiência do padrão WebP sobre o JPEG tradicional: para atingir um mesmo patamar de fidelidade matemática (como a faixa de excelente qualidade, onde PSNR > 40 dB), o codificador WebP gera arquivos significativamente menores. Esse comportamento traduz o impacto prático da evolução dos algoritmos na otimização de sistemas de transmissão e armazenamento digital.

```
%%writefile tmp/fig_05_formatos_comparacao.cpp
#define MM_OUT "tmp/fig_05_formatos_comparacao.png"
//| label: fig-05-formatos-comparacao
//| fig-cap: "Curva taxa-distorção: PSNR vs tamanho de arquivo para JPEG, WebP e PNG aplicada
à imagem do *Cameraman*."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
#include <filesystem>
#include <vector>
#include <string>
#include <iostream>
#include "morph.hpp"

int main() {
    // Cria diretório tmp se não existir
    std::filesystem::create_directories("tmp");

    // Lê e redimensiona a imagem
```

```
    cv::Mat img = cv::imread("imagens/cameraman.png", cv::IMREAD_GRAYSCALE);
    cv::Mat src;
    cv::resize(img, src, cv::Size(256, 256));

    // Converte para mm::Image para compatibilidade
    mm::Image src_mm(src);

    std::vector<double> jpeg_kb, jpeg_psnr;
    std::vector<int> jpeg_qualities = {10, 20, 30, 40, 50, 60, 70, 80, 90, 95};
    for (int q : jpeg_qualities) {
        std::string p = "tmp/fmt_q" + std::to_string(q) + ".jpg";
        std::vector<int> params = {cv::IMWRITE_JPEG_QUALITY, q};
        cv::imwrite(p, src, params);
        cv::Mat rec = cv::imread(p, cv::IMREAD_GRAYSCALE);
        double file_size = (double)std::filesystem::file_size(p) / 1024.0;
        jpeg_kb.push_back(file_size);
        jpeg_psnr.push_back(mm::psnr(src, rec));
    }

    std::vector<double> webp_kb, webp_psnr;
    std::vector<int> webp_qualities = {30, 50, 70, 85, 95};
    for (int q : webp_qualities) {
        std::string p = "tmp/fmt_w" + std::to_string(q) + ".webp";
        std::vector<int> params = {cv::IMWRITE_WEBP_QUALITY, q};
        cv::imwrite(p, src, params);
        cv::Mat rec = cv::imread(p, cv::IMREAD_GRAYSCALE);
        double file_size = (double)std::filesystem::file_size(p) / 1024.0;
        webp_kb.push_back(file_size);
        webp_psnr.push_back(mm::psnr(src, rec));
    }

    std::string p_png = "tmp/fmt.png";
    std::vector<int> png_params = {cv::IMWRITE_PNG_COMPRESSION, 9};
    cv::imwrite(p_png, src, png_params);
    double png_kb = (double)std::filesystem::file_size(p_png) / 1024.0;

    std::string title = "Curva Taxa-Distorcao (PNG sem perda: " +
                        std::to_string(png_kb) + " KB)";

    // Converte para std::vector<std::vector<double>> como esperado pelo lineChart
    std::vector<std::vector<double>> xs = {jpeg_kb, webp_kb};
    std::vector<std::vector<double>> ys = {jpeg_psnr, webp_psnr};

    mm::Image chart = mm::lineChart(
        xs, ys,
        {"JPEG", "WebP"},
        {},  // colors vazio
        title,
        "Tamanho do arquivo (KB)", "PSNR (dB)"
    );

    std::vector<mm::Image> display = {chart};
    mm::show(display, MM_OUT, {"JPEG x WebP x PNG"}, 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)")
```

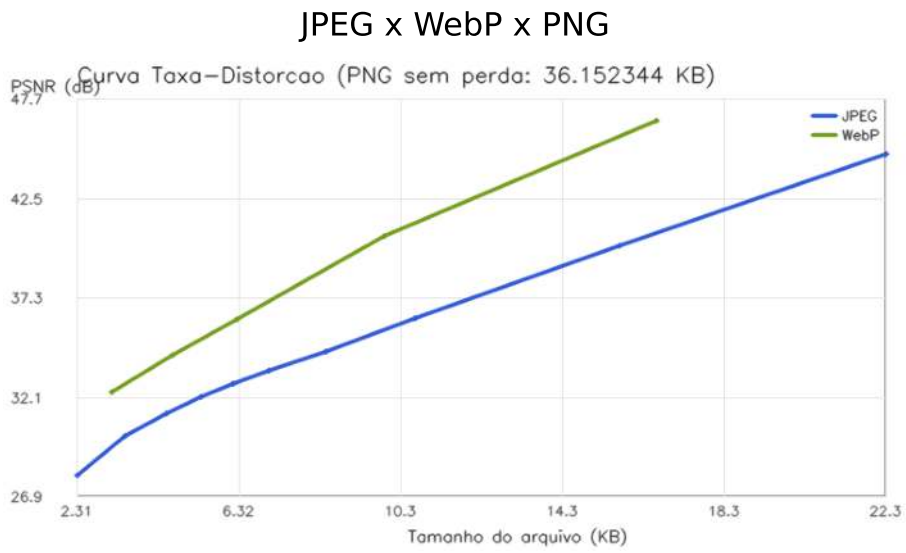


Figura 5.28: Curva taxa-distorção: PSNR vs tamanho de arquivo para JPEG, WebP e PNG aplicada à imagem do *Cameraman*.

i Tamanho original da imagem

A imagem *Cameraman* (256 × 256 pixels em escala de cinza) ocupa **64 KB** em formato bruto (sem compressão). Como referência, o PNG *lossless* comprime esse volume para **36,2 KB** — evidenciando que a compressão sem perdas já reduz significativamente o armazenamento para imagens com regiões homogêneas. Em contrapartida, os formatos com perda (JPEG e WebP) atingem tamanhos ainda menores: o JPEG com qualidade 95 ocupa 22,3 KB (PSNR 45 dB), enquanto o WebP com qualidade 90 atinge 12,5 KB com PSNR equivalente, demonstrando sua superioridade em eficiência de compressão.

5.8.4.2 Mapeamento Espacial de Erros e Correlação Perceptual

Embora o PSNR ofereça um indicativo numérico rápido, métricas globais falham em discriminar como a perda de informação se distribui geometricamente sobre a imagem. A Figura 5.29 soluciona essa limitação ao associar as reconstruções em diferentes qualidades aos seus respectivos mapas de erro absoluto e ao SSIM.

Os mapas residuais — obtidos pela diferença absoluta normalizada entre a imagem original e a comprimida — revelam a assinatura espacial intrínseca de cada arquitetura de codificação:

- **Em altas qualidades ($Q = 95$ a $Q = 75$):** As distorções concentram-se predominantemente ao redor de transições abruptas de intensidade (bordas), fruto do espelhamento espectral decorrente do descarte de altas frequências. O índice SSIM permanece próximo à unidade, atestando a integridade das estruturas originais.
- **Em qualidades agressivas ($Q = 50$ a $Q = 25$):** O erro assume uma estrutura de malha ortogonal regularizada. Esse padrão geométrico evidencia o surgimento dos **artefatos de bloco** (*blocking artifacts*), indicando que a quantização severa corrompeu a correlação espacial entre blocos adjacentes de 8×8 pixels.

O SSIM captura essa degradação morfológica de forma muito mais sensível que o PSNR, penalizando o escore final à medida que a organização estrutural e as texturas finas — às quais o sistema visual humano é altamente responsivo — são eliminadas pelo codificador.

```
%%writefile tmp/fig_05_ssim_artefatos.cpp
#define MM_OUT "tmp/fig_05_ssim_artefatos.png"
//| label: fig-05-ssim-artefatos
//| fig-cap: "Análise espacial de degradação: imagens reconstruídas e respectivos mapas de
erro absoluto normalizados para diferentes fatores de qualidade JPEG."
//| echo: true
//| output: true

#include <opencv2/opencv.hpp>
#include <cstdio>
#include <string>
#include <vector>
#include "morph.hpp"

// Implementação padrão de SSIM com janela gaussiana (Wang et al.)
static 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_64F);
    img2.convertTo(I2, CV_64F);

    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_sq = mu1.mul(mu1);
    cv::Mat mu2_sq = mu2.mul(mu2);
```

```
    cv::Mat mu1_mu2 = mu1.mul(mu2);

    cv::Mat sigma1_sq, sigma2_sq, sigma12;
    cv::GaussianBlur(I1.mul(I1), sigma1_sq, cv::Size(11, 11), 1.5);
    sigma1_sq -= mu1_sq;
    cv::GaussianBlur(I2.mul(I2), sigma2_sq, cv::Size(11, 11), 1.5);
    sigma2_sq -= mu2_sq;
    cv::GaussianBlur(I1.mul(I2), sigma12, cv::Size(11, 11), 1.5);
    sigma12 -= mu1_mu2;

    cv::Mat ssim_map;
    cv::Mat t1 = (2 * mu1_mu2 + C1).mul(2 * sigma12 + C2);
    cv::Mat t2 = (mu1_sq + mu2_sq + C1).mul(sigma1_sq + sigma2_sq + C2);
    cv::divide(t1, t2, ssim_map);
    return cv::mean(ssim_map)[0];
}

int main() {
    // Cameraman (asset do capítulo), 256x256 - mesma imagem da trilha py.
    cv::Mat img_gray_tmp = cv::imread("imagens/cameraman.png", cv::IMREAD_GRAYSCALE);
    cv::resize(img_gray_tmp, img_gray_tmp, cv::Size(256, 256));
    cv::Mat img_gray = img_gray_tmp; // mm::gray já é 1 canal; aqui já veio 1 canal

    std::vector<cv::Mat> imgs_ssim;
    std::vector<std::string> titles_ssim;
    imgs_ssim.push_back(img_gray);
    titles_ssim.push_back("Original");

    for (int q : {25, 50, 75, 95}) {
        cv::Mat rec = mm::jpegCompress(img_gray, q);  // DCT 8x8 -> quant -> IDCT
        double psnr_v = mm::psnr(img_gray, rec);
        double ssim_v = compute_ssim(img_gray, rec);

        cv::Mat diff64, diff_vis;
        cv::absdiff(img_gray, rec, diff64);
        diff64.convertTo(diff64, CV_64F);
        cv::normalize(diff64, diff_vis, 0, 255, cv::NORM_MINMAX, CV_8U);

        imgs_ssim.push_back(rec);
        imgs_ssim.push_back(diff_vis);

        char buf1[128], buf2[128];
        snprintf(buf1, sizeof(buf1), "Q=%d (PSNR=%.1fdB | SSIM=%.3f)", q, psnr_v, ssim_v);
        snprintf(buf2, sizeof(buf2), "Mapa de erro (Q=%d) - bordas e blocagem", q);
        titles_ssim.push_back(buf1);
        titles_ssim.push_back(buf2);
    }

    // Reconstrução dos mm::Image para exibição
    std::vector<mm::Image> out_imgs;
    for (auto& m : imgs_ssim) out_imgs.emplace_back(m);
    mm::show(out_imgs, 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] Mapa de erro (Q=25) - bordas e blocagem
[4] Q=50 (PSNR=32.8dB | SSIM=0.905)
[5] Mapa de erro (Q=50) - bordas e blocagem
[6] Q=75 (PSNR=35.2dB | SSIM=0.938)
[7] Mapa de erro (Q=75) - bordas e blocagem
[8] Q=95 (PSNR=44.8dB | SSIM=0.990)
[9] Mapa de erro (Q=95) - bordas e blocagem
```

```
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)")
```

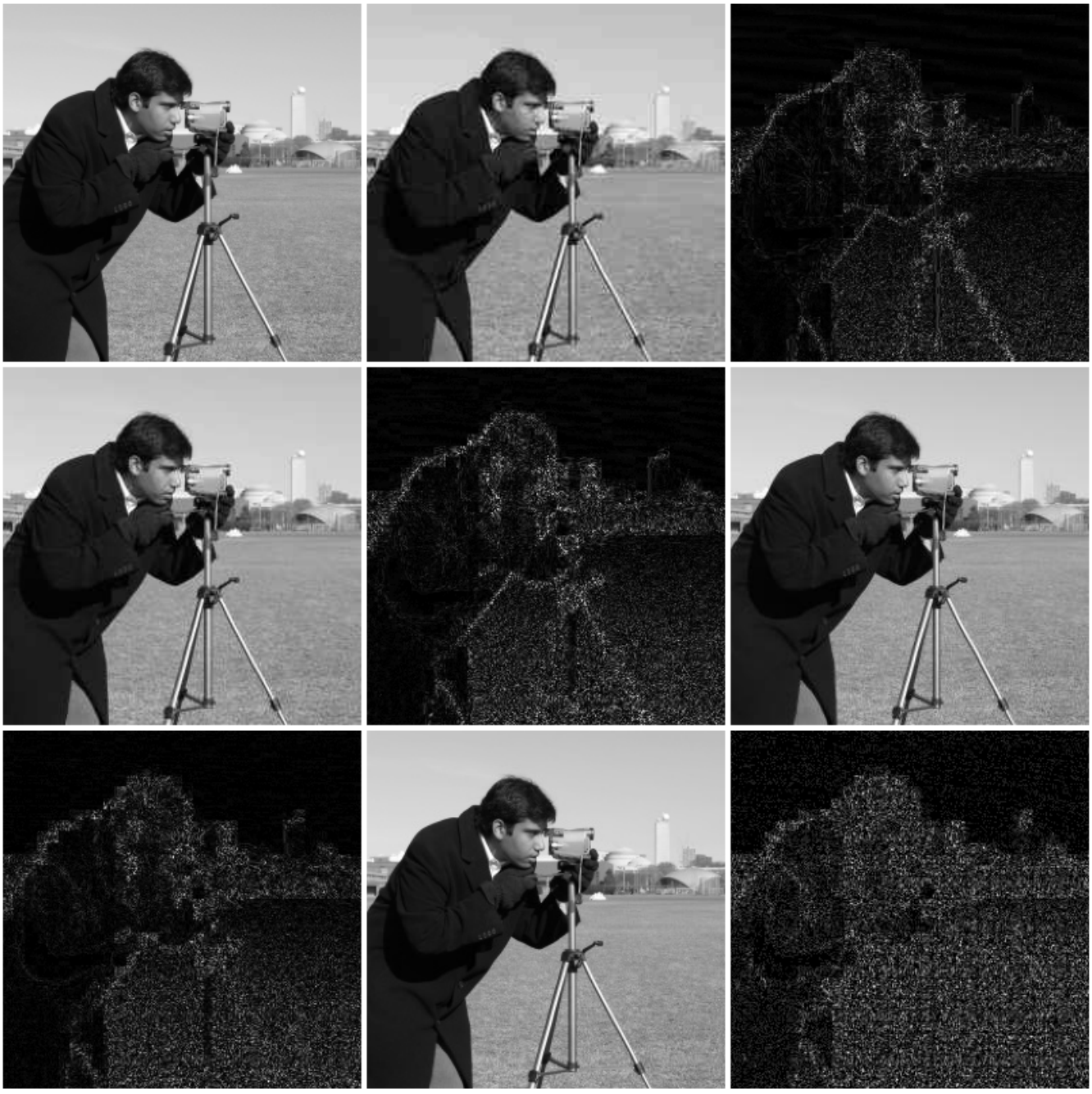


Figura 5.29: Análise espacial de degradação: imagens reconstruídas e respectivos mapas de erro absoluto normalizados para diferentes fatores de qualidade JPEG.

i Interpretando os mapas de erro

Os mapas de erro apresentados foram **normalizados individualmente** (`cv2.NORM_MINMAX`) para maximizar o contraste visual e revelar a estrutura espacial das distorções. Isso significa que:

- Em **Q=95**, o erro absoluto é da ordem de **0.5–1.5 níveis de cinza** (imperceptível visualmente), mas a normalização o amplifica para preto-branco para evidenciar sua localização em bordas e transições.
- Em **Q=25**, o erro absoluto é **10–20 vezes maior** (5–15 níveis de cinza), mas a normalização também o leva ao mesmo intervalo [0, 255].

Portanto, **a intensidade do branco nos mapas NÃO é comparável entre diferentes qualidades** — os mapas servem apenas para revelar a **assinatura espacial** do erro (bordas vs blocos), não sua

magnitude. A magnitude correta é dada pelos valores de PSNR e SSIM, que mostram claramente que Q=95 tem erro muito menor que Q=25.

Síntese — Compressão JPEG

O processo de compressão no padrão JPEG baseia-se na aplicação combinada de transformações espaciais, perceptuais e estatísticas para reduzir as redundâncias de uma imagem. A Tabela 5.9 resume o papel de cada etapa no *pipeline* e seu respectivo impacto na redução de dados.

Tabela 5.9: Síntese das etapas do *pipeline* de compressão JPEG e seus respectivos impactos.

Etapa	Operação Analítica	Mecanismo de Ganho / Compressão
Conversão YC_bC_r	Isolamento dos canais de luminância e crominância.	Modela a percepção do SVH, permitindo tratar cor e brilho de forma independente.
Subamostragem 4:2:0	Redução da resolução espacial dos canais de cor (C_b e C_r).	Elimina aproximadamente 50% dos dados brutos com impacto visual mínimo.
DCT 8×8	Mapeamento do domínio espacial para o domínio de frequências espaciais.	Compactação de energia, concentrando a informação vital nos primeiros coeficientes.
Quantização Linear	Divisão inteira dos coeficientes por uma matriz de ponderação $Q(u, v)$.	Principal fonte de compressão com perda; elimina altas frequências imperceptíveis.
Codificação Entrópica	Aplicação de algoritmos RLE e codificação de Huffman.	Compressão estatística sem perda, otimizada pelas longas corridas de coeficientes nulos.

Artefatos de Degradação Característicos

A aplicação de taxas de compressão excessivamente agressivas (fatores de qualidade reduzidos) introduz distorções previsíveis na imagem reconstruída, decorrentes das limitações matemáticas do modelo:

- **Artefatos de bloco (*blocking artifacts*):** Descontinuidades geométricas visíveis nas fronteiras dos blocos de 8×8 pixels, causadas pela perda de correlação espacial após a quantização severa das componentes AC.
- **Efeito de espalhamento (*ringing*):** Oscilações fantasmas ou distorções de "fumaça" ao redor de bordas nítidas e de alto contraste, provocadas pela eliminação abrupta de harmônicos de alta frequência necessários para reconstruir funções degrau.
- **Perda de textura fina:** Atenuação de detalhes de alta frequência e baixo contraste (como gramados, tecidos ou porosidade), fazendo com que regiões originalmente texturizadas assumam um aspecto excessivamente liso ou homogeneizado.

5.9 Aplicação Prática: Remoção de Ruído por Filtragem Híbrida

Reunindo as técnicas consolidadas ao longo deste capítulo, apresenta-se um *pipeline* completo de **restauração de imagens** que combina a análise espectral no domínio da frequência com a filtragem adaptativa no domínio espacial. O objetivo é atenuar um ruído misto (composto por degradação Gaussiana e interferência periódica) preservando ao máximo os detalhes estruturais da imagem original.

$$\text{Imagem Ruidosa} \xrightarrow{\text{FFT2}} \xrightarrow{\text{Filtro Notch Gaussiano}} \xrightarrow{\text{IFFT2}} \xrightarrow{\text{Filtro Bilateral}} \text{Imagem Restaurada}$$

Avaliação Complementar: PSNR vs. SSIM

O par de métricas estatísticas PSNR e SSIM fornece uma avaliação qualitativa e morfológica complementar do processo de restauração:

- **PSNR:** Penaliza uniformemente o desvio quadrático médio pixel a pixel.
- **SSIM:** Avalia a preservação de estruturas locais perceptualmente relevantes (luminância, contraste e contornos).

Na prática, existe um compromisso analítico (*trade-off*) entre **redução de ruído** e **preservação de detalhes**: filtros espaciais excessivamente agressivos atenuam bem o ruído de alta frequência, mas degradam texturas finas e suavizam bordas nítidas — o que **reduz simultaneamente** tanto o PSNR quanto o SSIM em relação à imagem original. O desafio do projeto de filtros é encontrar o ponto de equilíbrio que maximize ambas as métricas, garantindo uma restauração fiel e visualmente agradável.

5.9.1 Análise de Desempenho e Conclusão do Capítulo

Os resultados numéricos e visuais gerados pela Figura 5.30 demonstram a relevância prática de associar diferentes domínios de processamento. A inserção simultânea de ruído periódico e estocástico corrompe as propriedades morfológicas do sinal, reduzindo severamente os índices de similaridade e a relação sinal-ruído da imagem de referência.

O isolamento e a supressão dos picos harmônicos no domínio da frequência por meio da máscara *notch* removem as franjas de interferência senoidais espalhadas sobre o espaço bi-dimensional. Como evidenciado nos dados impressos da Figura 5.30, essa filtragem cirúrgica promove um salto imediato e substancial na métrica PSNR. Contudo, o ruído Gaussiano de alta frequência permanece ativo de forma homogênea no espectro, exigindo uma abordagem complementar.

A restauração final é consolidada no domínio espacial com a introdução do filtro bilateral. Diferentemente de operadores passa-baixas convencionais (como o Gaussiano ou de média), que suavizariam indiscriminadamente o ruído e os contornos estruturais, a filtragem bilateral calcula pesos ponderados pela proximidade geométrica e pela diferença de intensidade radiométrica. Esse comportamento adaptativo atenua as flutuações estocásticas remanescentes nas regiões de transição suave e preserva a nitidez das bordas espaciais.

A convergência de ambas as abordagens resulta em uma **melhoria substancial e simultânea** do PSNR e do SSIM em relação à imagem ruidosa — embora os valores finais permaneçam inferiores aos da imagem original (PSNR $= \infty$, SSIM $= 1{,}0$), devido à perda inevitável de informações espectrais e texturais durante os processos de filtragem. A atenuação suave (gaussiana) dos picos no espectro evita artefatos de *ringing*, enquanto o filtro bilateral elimina o ruído estocástico residual sem comprometer a nitidez das bordas. Os resultados comprovam a eficácia e a complementaridade prática das ferramentas de análise de frequência apresentadas neste capítulo, demonstrando que a filtragem híbrida (frequência + espacial) é superior a qualquer abordagem isolada para a restauração de imagens degradadas por ruído misto.

```
%%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 <vector>
#include <string>
#include <cmath>
#include <random>
```

```cpp
#include <iostream>
#include "morph.hpp"

int main() {
    // Cameraman (asset do capítulo), 256x256 - mesma imagem da trilha py.
    mm::Image img_gray = mm::gray(mm::read("imagens/cameraman.png"));
    cv::Mat img_gray_cv = img_gray;
    cv::resize(img_gray_cv, img_gray_cv, cv::Size(256, 256));
    img_gray = mm::Image(img_gray_cv);
    int h_img = img_gray.h;
    int w_img = img_gray.w;

    //   1. Ruído misto: gaussiano + periódico
    std::mt19937 gen(42);
    std::normal_distribution<double> dist(0.0, 15.0);

    cv::Mat ruido_gauss(h_img, w_img, CV_64F);
    for (int y = 0; y < h_img; y++) {
        for (int x = 0; x < w_img; x++) {
            ruido_gauss.at<double>(y, x) = dist(gen);
        }
    }

    double u0 = 15.0, v0 = 10.0;
    cv::Mat ruido_period(h_img, w_img, CV_64F);
    for (int y = 0; y < h_img; y++) {
        for (int x = 0; x < w_img; x++) {
            ruido_period.at<double>(y, x) = 30.0 * std::sin(2.0 * M_PI * (u0 * x / w_img + v0
* y / h_img));
        }
    }

    cv::Mat img_gray_float;
    img_gray_cv.convertTo(img_gray_float, CV_64F);
    cv::Mat img_noisy_float = img_gray_float + ruido_gauss + ruido_period;
    cv::Mat img_noisy_clipped;
    cv::threshold(img_noisy_float, img_noisy_clipped, 0, 255, cv::THRESH_TOZERO);
    cv::threshold(img_noisy_clipped, img_noisy_clipped, 255, 255, cv::THRESH_TRUNC);
    cv::Mat img_noisy_8u;
    img_noisy_clipped.convertTo(img_noisy_8u, CV_8U);
    mm::Image img_noisy(img_noisy_8u);

    //   2. Espectro (log-magnitude) da imagem ruidosa
    mm::Image mag_n = mm::spectrumMag(img_noisy);

    //   3. Máscara notch gaussiana nos 4 picos periódicos
    cv::Mat mascara_notch = cv::Mat::ones(h_img, w_img, CV_64F);
    int cy0 = h_img / 2, cx0 = w_img / 2;

    auto suprimir_pico_gaussiano = [&](cv::Mat& mask, int cy, int cx, double sigma) {
        for (int y = 0; y < h_img; y++) {
            for (int x = 0; x < w_img; x++) {
                double dx = x - cx;
                double dy = y - cy;
                double notch = std::exp(-(dy * dy + dx * dx) / (2.0 * sigma * sigma));
                mask.at<double>(y, x) *= (1.0 - notch);
            }
        }
    };

    int dys[] = {(int)v0, -(int)v0, (int)v0, -(int)v0};
```

```
    int dxs[] = {(int)u0, -(int)u0, -(int)u0, (int)u0};
    for (int i = 0; i < 4; i++) {
        suprimir_pico_gaussiano(mascara_notch, cy0 + dys[i], cx0 + dxs[i], 3.0);
    }

    //   4. Filtragem: notch no domínio da frequência + bilateral
    mm::Image img_notch = mm::freqFilter(img_noisy, mascara_notch);

    cv::Mat img_notch_cv = img_notch;
    cv::Mat img_den_cv;
    cv::bilateralFilter(img_notch_cv, img_den_cv, 7, 25, 7);
    mm::Image img_den(img_den_cv);

    cv::Mat mascara_vis_float = mascara_notch * 255.0;
    cv::Mat mascara_vis_8u;
    mascara_vis_float.convertTo(mascara_vis_8u, CV_8U);
    mm::Image mascara_vis(mascara_vis_8u);

    double psnr_n = mm::psnr(img_gray, img_noisy);
    double psnr_no = mm::psnr(img_gray, img_notch);
    double psnr_d = mm::psnr(img_gray, img_den);
    char buffer[200];
    snprintf(buffer, sizeof(buffer), "Ruidosa: PSNR=%.2f dB | Apos notch: %.2f dB |
Notch+bilateral: %.2f dB", psnr_n, psnr_no, psnr_d);
    std::cout << buffer << std::endl;

    // Preparar títulos com PSNR
    char titulo_ruidosa[100], titulo_notch[100], titulo_den[100];
    snprintf(titulo_ruidosa, sizeof(titulo_ruidosa), "Ruidosa (PSNR=%.1f dB)", psnr_n);
    snprintf(titulo_notch, sizeof(titulo_notch), "Apos notch (PSNR=%.1f dB)", psnr_no);
    snprintf(titulo_den, sizeof(titulo_den), "Notch + bilateral (PSNR=%.1f dB)", psnr_d);

    std::vector<mm::Image> imagens = {img_gray, img_noisy, mag_n, mascara_vis, img_notch,
img_den};
    std::vector<std::string> titulos = {
        "Original",
        titulo_ruidosa,
        "Espectro (picos visiveis)",
        "Mascara notch (gaussiana)",
        titulo_notch,
        titulo_den
    };

    mm::show(imagens, MM_OUT, titulos, 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=20.2 dB)
[3] Espectro (picos visiveis)
[4] Mascara notch (gaussiana)
[5] Apos notch (PSNR=22.4 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)")
```

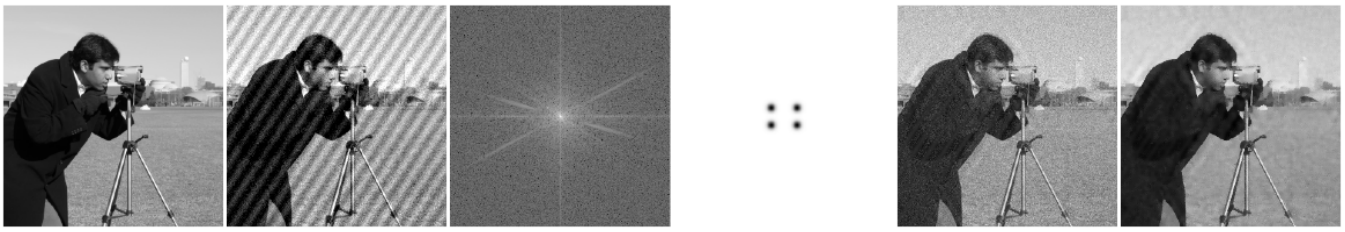


Figura 5.30: *Pipeline* completo de remoção de ruído misto: (1) adição de ruído gaussiano e periódico; (2) identificação de picos de interferência no espectro de frequências; (3) aplicação de máscara *notch* com atenuação gaussiana suave; (4) pós-processamento via filtro bilateral para eliminação do ruído estocástico residual.

5.10 Resumo do Capítulo

A transição do **domínio espacial** para o **domínio da frequência** revela a distribuição espectral de energia da imagem, estabelecendo a base analítica para a filtragem avançada, restauração e compressão de dados. A articulação estrutural desses conceitos é sintetizada no mapa conceitual da Figura 5.31.

Fundamentos Essenciais

- **DFT e Percepção Visual:** O espectro decompõe a imagem em componentes harmônicas. A **fase** retém a inteligibilidade geométrica da cena e a localização de contornos, enquanto a **magnitude** dita a distribuição de contraste e amplitudes globais.
- **Eficiência Algorítmica:** O Teorema da Convolução viabiliza o processamento de máscaras de grande escala no domínio da frequência via FFT, reduzindo a complexidade computacional assintótica de $O(N^2K^2)$ no espaço para $O(N^2 \log N)$.

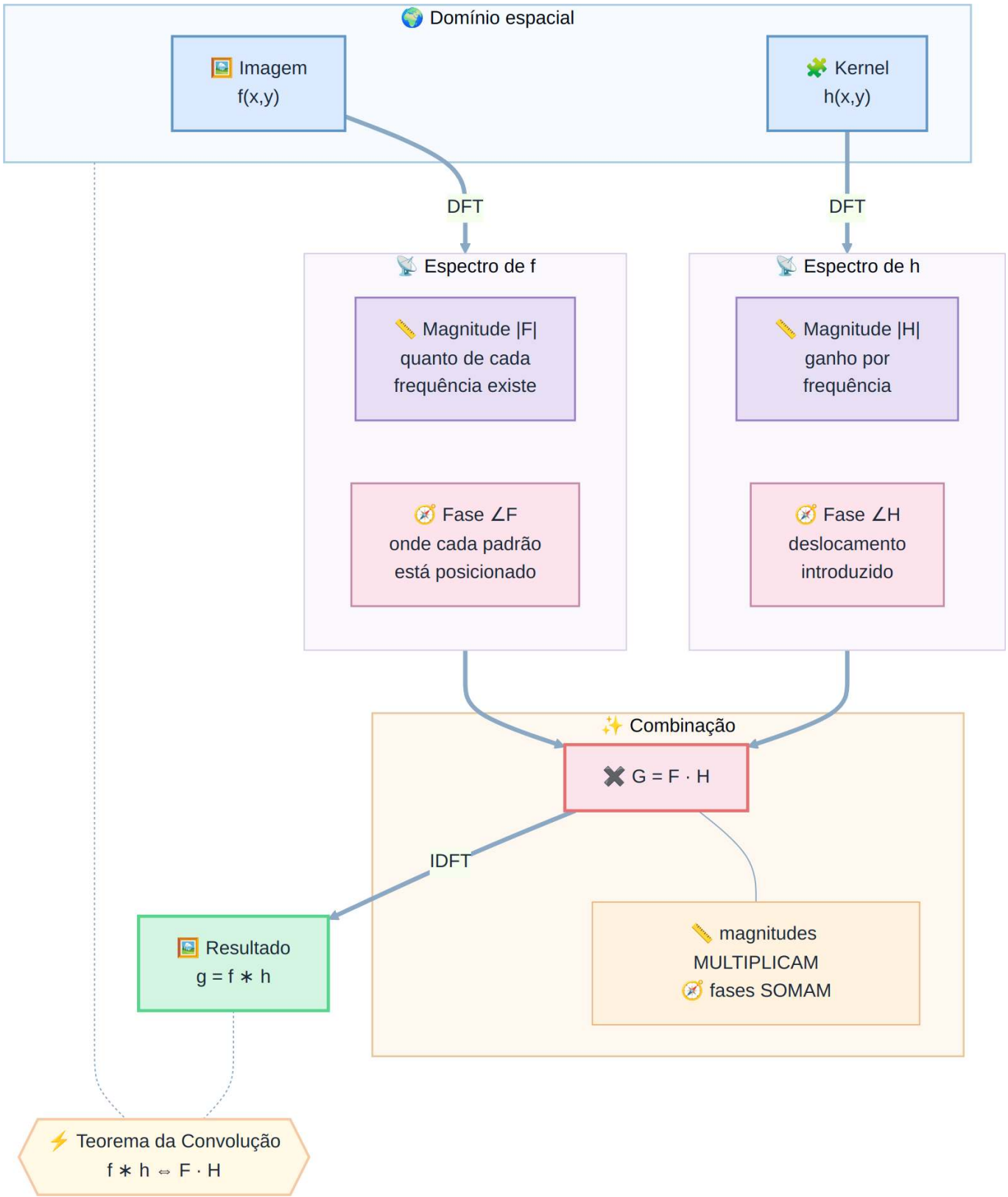


Figura 5.31: Mapa conceitual das transformações e propriedades no domínio da frequência.

- **Fenômeno de *Ringing*:** Cortes abruptos no espectro (Filtros Ideais) geram oscilações espaciais indesejadas (fenômeno de Gibbs). A atenuação suave por filtros de **Butterworth** ou **Gaussianos** elimina essas descontinuidades.
- **Análise Multirresolução via *Wavelets*:** Superando o caráter puramente global de Fourier, a DWT captura a frequência e a localização espacial simultaneamente, fundamentando o padrão JPEG 2000 e subsidiando representações hierárquicas análogas às extrações de feições em Redes Neurais Convolucionais (CNNs).
- **Compressão Perceptual (DCT):** O *pipeline* JPEG explora as limitações de contraste do sistema visual humano em altas frequências espaciais. A DCT isola a energia de blocos 8×8, permitindo que a quantização descarte coeficientes AC de detalhes finos sem prejuízo perceptual severo.

Próximos Passos: O **Capítulo 6** inaugura a Parte II da obra, aplicando as ferramentas de processamento de imagens na resolução de problemas reais de inspeção industrial. Serão exploradas técnicas de **segmentação e análise de formas** para a detecção automática de falhas em linhas de produção — desde a identificação de defeitos superficiais em peças até a leitura *QRCode* em provas, consolidando a ponte entre a teoria apresentada na Parte I e as demandas práticas da visão computacional.

5.11 🤖 Uso do Gemini Notebook como Tutor Complementar

Nesta edição, incentiva-se o uso da plataforma **Gemini Notebook** como ferramenta complementar de aprendizagem — **não como substituta** da leitura atenta, da resolução de exercícios ou da experimentação prática. Baseado em arquiteturas de inteligência artificial, o sistema utiliza exclusivamente o material didático e os documentos fornecidos pelo autor como base de conhecimento, assegurando que as respostas geradas estejam conceitualmente alinhadas ao conteúdo programático e à abordagem pedagógica adotada ao longo desta obra.

Acesso ao Tutor Inteligente

🚀 ACESSAR Gemini Notebook: CAPÍTULO 05

🌐 Idioma e Linguagem de Programação

O projeto deste capítulo no Gemini Notebook foi construído apenas com o texto em **português** e os exemplos de código em **Python**. Se você está estudando pela edição em inglês ou francês, ou acompanhando a trilha em C++, as respostas do tutor podem não corresponder exatamente à versão que você está lendo.

Diretrizes sobre o Conteúdo Gerado por Inteligência Artificial

Embora as ferramentas de inteligência artificial constituam aliados eficientes no processo de aprendizagem e revisão, o conteúdo gerado está sujeito a inconsistências ou imprecisões técnicas. Desse modo, é indispensável a consulta sistemática a livros-texto, artigos científicos e fontes acadêmicas indexadas para a validação rigorosa das informações. Recomenda-se veementemente a execução e a modificação dos exemplos práticos em Python fornecidos neste capítulo como método primário de verificação experimental dos resultados.

5.12 Lista de Exercícios

1. **(10%) Implementação Direta da DFT 2D:** Implemente analiticamente a Transformada Discreta de Fourier 2D (DFT) sem o auxílio de funções nativas de bibliotecas (como `np.fft.fft2`), utilizando estritamente a formulação matemática definida na Equação 5.1 para uma matriz de dimensões 16×16. Realize a validação numérica comparando os coeficientes gerados com os resultados da função `np.fft.fft2`, certificando-se de que o desvio absoluto máximo seja inferior a 10^{-8}. Mensure os tempos

de execução de ambos os métodos e apresente uma justificativa teórica para a disparidade observada em termos de complexidade assintótica.

2. **(15%) Supressão de Ruído Periódico:** Adicione interferências senoidais com frequências espaciais $(u_0, v_0) \in \{(5, 10), (20, 5), (30, 30)\}$ à imagem de teste do *Cameraman*. Para cada cenário de degradação, projete uma máscara de filtragem *notch* específica no domínio da frequência para isolar e atenuar os picos harmônicos indesejados. Avalie quantitativamente a eficácia do processo de restauração por meio do cálculo das métricas de PSNR e SSIM. Discuta analiticamente o compromisso (*trade-off*) entre a atenuação do ruído senoidal e a indesejada atenuação de feições estruturais legítimas da imagem.

3. **(15%) Análise Comparativa de Operadores Passa-Baixa:** Realize um estudo comparativo entre os filtros passa-baixa Ideal, Gaussiano e Butterworth (com ordens harmônicas $n = 1, 2, 4$), parametrizados com frequências de corte $D_0 = 20, 40, 60$ pixels. Para cada combinação estrutural, calcule os índices PSNR e SSIM da imagem resultante frente ao sinal original de referência. Organize os dados quantitativos em uma tabela estruturada e plote os gráficos unidimensionais das funções de transferência correspondentes ao longo do perfil horizontal $H(u, 0)$.

4. **(15%) Banco de Filtros Multirresolução de Haar:** Desenvolva um script para executar manualmente a decomposição *wavelet* discreta 2D de primeiro nível utilizando a família Haar. O algoritmo deve calcular os coeficientes dos filtros correspondentes passa-baixa (h) e passa-alta (g), aplicando-os de forma separável sobre as linhas e colunas da matriz, seguidos pela operação de decimação (subamostragem espacial por um fator de 2). Valide numericamente a exatidão da sua implementação confrontando as subbandas obtidas com a saída da função `pywt.dwt2(img, 'haar')`.

5. **(15%) Compressão Esparsa por Limiarização Wavelet:** Aplique a técnica de filtragem por limiarização abrupta (*hard thresholding*) sobre os coeficientes de detalhe da decomposição *wavelet*, adotando os limiares numéricos $T \in \{5, 10, 20, 40, 80\}$ para as famílias Haar, Daubechies (`db4`) e Symlets (`sym4`). Após realizar o processo de síntese por meio da transformada inversa (`pywt.waverec2`), compute os valores de PSNR e SSIM de cada imagem reconstruída. Identifique e justifique qual combinação de família *wavelet* e limiar T maximiza a similaridade estrutural.

6. **(15%) Construção de Codificador JPEG Simplificado:** Implemente o pipeline completo de compressão de dados simulando o padrão JPEG. O fluxo deve englobar: conversão espacial $RGB \rightarrow YC_bC_r$, subamostragem cromática na proporção 4:2:0, segmentação da luminância em blocos disjuntos de 8×8 pixels, aplicação da DCT-II 2D ortogonal, e quantização linear baseada na matriz normalizada de luminância escalada por fatores de qualidade desejados. Realize a decodificação inversa e compare quantitativamente as reconstruções com os arquivos gerados pela função `cv2.imencode` para os fatores de qualidade de 20, 50 e 80.

7. **(15%) Análise Perceptual em Conteúdos Heterogêneos:** Desenvolva uma imagem sintética composta por três regiões distintas e de características espectrais contrastantes: uma textura fotográfica complexa (representando altas frequências estocásticas), uma área de texto vetorizado com bordas nítidas (representando transições degrau puras) e um gradiente linear contínuo (representando baixas frequências homogêneas). Submeta essa imagem mista aos processos de compressão sob os formatos JPEG, PNG e WebP. Avalie e interprete os resultados correlacionando o tamanho final do arquivo em disco às métricas PSNR e SSIM obtidas, justificando qual formato exibe o melhor desempenho para sinais de natureza heterogênea e por que essa vantagem ocorre em termos de compactação de energia e preservação perceptual.

Referências do Capítulo

A fundamentação teórica e o desenvolvimento analítico dos conceitos abordados neste capítulo fundamentam-se nas seguintes obras de referência:

- **Gonzalez (2018)** — Formulações clássicas de Transformadas Discretas de Fourier 2D (DFT), projeto de filtros analíticos no domínio da frequência, Transformada Discreta de Cossenos (DCT) e princípios fundamentais de sistemas de compressão de imagens.

- **Oppenheim (2010)** — Teoria formal de sinais e sistemas aplicados no domínio discreto, cobrindo as propriedades matemáticas da DFT e a modelagem analítica do Teorema da Convolução.
- **Mallat (1999)** — Fundamentação matemática da teoria de *wavelets*, formalização da análise multirresolução (MRA) e arquitetura de bancos de filtros diádicos.
- **Wallace (1991)** — Especificação original e aspectos de engenharia do padrão de compressão ISO/IEC JPEG, com ênfase nos critérios psicovisuais para o projeto de matrizes de quantização DCT.
- **Szeliski (2022)** — Modelagem computacional e caracterização de métricas modernas de fidelidade e qualidade perceptual (PSNR e SSIM), bem como a análise comparativa de formatos de imagem rasterizados de alto desempenho.

5.13 💻 Parte Prática com Exercícios de Programação

A presente lista de exercícios de programação (EP) consolida as formulações teóricas apresentadas ao longo do Capítulo 5 — Transformadas e Compressão — por meio de uma trilha prática aplicada. Os exercícios são estruturados a partir de matrizes de dimensões reduzidas, viabilizando a validação analítica e a inspeção manual de cada coeficiente, mantendo a consistência metodológica adotada nos capítulos anteriores.

O encadeamento dos exercícios reproduz rigorosamente o fluxo conceitual do capítulo: inicia-se com a implementação explícita da Transformada Discreta de Fourier (DFT) a partir de sua definição matemática fundamental; avança-se para o projeto de filtros passa-baixa e máscaras *notch* no domínio da frequência; aplica-se a quantização de coeficientes (núcleo da compressão com perda); e conclui-se com a integração dessas etapas na construção de um *pipeline* de compressão JPEG simplificado e na análise perceptual de formatos de imagem.

! Diretrizes para a Resolução dos Exercícios de Programação

Em todos os exercícios deste capítulo, as coordenadas do **centro do espectro** (origem das frequências espaciais pós-aplicação do deslocamento `fftshift`) devem ser determinadas via divisão inteira. Para uma matriz com L linhas e C colunas, a componente de frequência nula localiza-se na posição:

$$(c_y, c_x) = \left(\left\lfloor \frac{L}{2} \right\rfloor, \left\lfloor \frac{C}{2} \right\rfloor \right)$$

Esta convenção é rigorosamente idêntica à adotada pela função `np.fft.fftshift`. Ademais, em todas as etapas que exijam discretização ou arredondamento numérico (seja na quantização de coeficientes AC ou na reconstrução final de pixels), deve-se empregar o arredondamento padrão para o inteiro mais próximo (*round half away from zero*), mitigando ambiguidades em valores com fração exatamente igual a 0.5.

🎯 Objetivo deste Caderno

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

Download

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

```
import os, 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
```

```
Ambiente pronto. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Executando os Testes

Para avaliar os testes, execute `TestSuite("EP05_01.extensão").run()` numa nova célula, trocando a extensão pela da linguagem usada (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). O sistema baixa os casos de teste do GitHub, executa o programa e calcula a nota automaticamente.

Para testar código Python diretamente, sem salvar arquivo, use `run_code(codigo)` passando o código como *string* numa variável `codigo`:

```
codigo = """
from morph import mm
# ... seu código aqui ...
"""
TestSuite("EP05_01").run_code(codigo)
```

5.13.1 EP05_01 🟢 Filtro Passa-Baixa Ideal por Distância no Espectro

Em um ***scanner* de documentos antigo**, o sensor capta papel amassado e textura de fibra junto com o texto — ruído de alta frequência que "polui" o espectro nas bordas. O técnico de manutenção não tem acesso à imagem original, apenas ao **espectro de magnitude já calculado** pelo software do *scanner*. Seu trabalho é simples e cirúrgico: manter apenas o **círculo central** de baixas frequências (a estrutura global do documento) e apagar tudo que estiver fora do raio D_0, eliminando a textura fina sem nem precisar tocar na imagem espacial.

Este é o **Filtro Passa-Baixa Ideal (LPFI)**: a operação espectral mais direta do capítulo, mas também a que melhor revela a anatomia de um espectro centrado.

5.13.1.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros L (linhas) e C (colunas) do espectro de magnitude — já fornecido **centrado** (equivalente à saída de `np.fft.fftshift`).
2. **Frequência de corte:** Ler o inteiro D_0.
3. **Dados:** Ler os valores inteiros da matriz de magnitude, linha a linha.
4. **Centro do espectro:** Calcular $(c_y, c_x) = (L\ //\ 2,\ C\ //\ 2)$.
5. **Distância:** Para cada posição (u, v), calcular

$$D(u,v) = \sqrt{(u-c_y)^2 + (v-c_x)^2}$$

6. **Máscara ideal:** Aplicar

$$H(u,v) = \begin{cases} 1, & D(u,v) \le D_0 \\ 0, & D(u,v) > D_0 \end{cases}$$

7. **Filtragem:** O valor de saída é $\text{mag}'(u,v) = \text{mag}(u,v) \cdot H(u,v)$.
8. **Saída:** Exibir a matriz filtrada com dimensões $L \times C$.

5.13.1.2 📌 Restrições Computacionais

- **Comparação não estrita:** o critério usa $D(u,v) \leq D_0$ (a fronteira pertence ao filtro, ou seja, é mantida).
- **Tipo:** todos os valores de entrada e saída são inteiros; a distância é calculada em ponto flutuante apenas internamente.
- **Sem arredondamento de magnitude:** como a entrada já é inteira e a máscara é binária (0 ou 1), a saída nunca precisa de arredondamento.

5.13.1.3 🧠 Fundamentação Teórica

Região	Distância ao centro	Efeito do filtro
Centro ($D \leq D_0$)	Baixas frequências	Preservadas — estrutura global mantida
Bordas ($D > D_0$)	Altas frequências	Zeradas — textura e ruído removidos
D_0 **pequeno**	—	Imagem reconstruída ficaria muito borrada
D_0 **grande**	—	Pouca filtragem; quase toda energia preservada

5.13.1.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linha 3: Inteiro D_0.
- Linhas seguintes: Elementos inteiros da matriz de magnitude (centrada).

Saída:

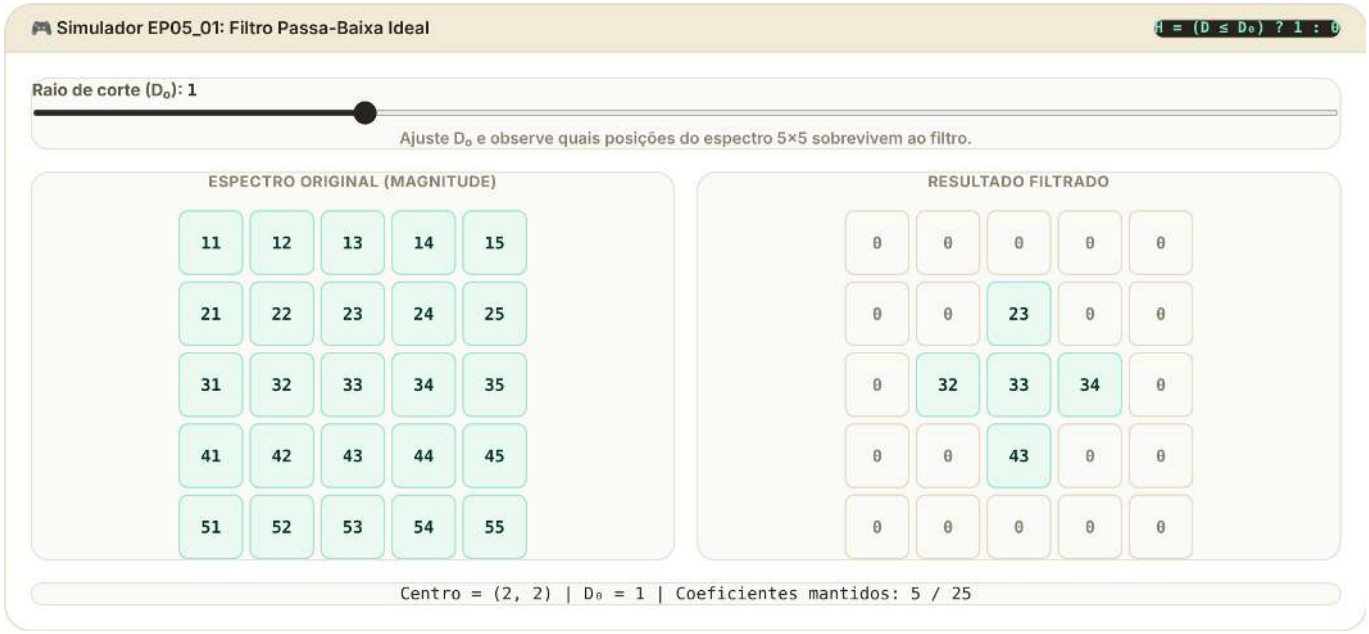
- Matriz filtrada em L linhas e C colunas, separados por espaço.

5.13.1.5 📌 Exemplos

Entrada	Saída	Observação
3 3 1 10 20 30 40 50 60 70 80 90	0 20 0 40 50 60 0 80 0	Centro $(1,1)$. Cantos têm $D = \sqrt{2} \approx 1.41 > 1$, logo são zerados; vizinhos ortogonais têm $D = 1 \leq 1$ e são mantidos.
1 3 0 5 9 7	0 9 0	$L = 1, C = 3$: centro em $(0,1)$. Apenas a própria posição central $(D = 0)$ sobrevive a $D_0 = 0$.

```
%%writefile EP05_01.cpp
// sua solução
```

```
Overwriting EP05_01.cpp
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap05/sim-ep0501.html

Figura 5.32: Simulador EP05_01: Filtro Passa-Baixa Ideal no Espectro

```
TestSuite("EP05_01.cpp").run()
```

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

5.13.2 EP05_02 🟡 Filtro *Notch*: Removendo Picos Periódicos

Uma câmera de **inspeção industrial** captura imagens de placas de circuito, mas a fonte de alimentação da linha de produção introduz uma **interferência elétrica periódica** — um padrão de listras quase imperceptível a olho nu, mas que aparece no espectro de Fourier como **pares de picos brilhantes** simetricamente posicionados em torno do centro. A equipe de visão computacional não pode reprocessar a captura: precisa **localizar e apagar cirurgicamente** esses pares de picos no espectro, preservando todo o resto da informação útil da imagem.

Esse é o papel do **filtro rejeita-banda *notch***: diferente do passa-baixa (que afeta uma região contínua), ele ataca **pontos específicos e seus simétricos**, deixando o restante do espectro intocado.

5.13.2.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros L (linhas) e C (colunas) do espectro de magnitude centrado.
2. **Dados:** Ler os valores inteiros da matriz de magnitude, linha a linha.
3. **Picos:** Ler o inteiro K (quantidade de pares de picos a remover).
4. **Para cada um dos K picos:** ler três inteiros Δv, Δu, r — deslocamento vertical, deslocamento horizontal e raio do *notch*.
5. **Centro do espectro:** $(c_y, c_x) = (L \,//\, 2,\ C \,//\, 2)$.
6. **Supressão simétrica:** para cada pico, zerar **todas** as posições (u, v) tais que a distância ao ponto $(c_y + \Delta v,\ c_x + \Delta u)$ for $\leq r$, **e também** todas as posições com distância $\leq r$ ao ponto simétrico $(c_y - \Delta v,\ c_x - \Delta u)$.
7. **Saída:** Exibir a matriz resultante com dimensões $L \times C$.

5.13.2.2 📌 Restrições Computacionais

- **Simetria obrigatória:** cada pico informado gera **dois** discos zerados (o ponto e seu simétrico em relação ao centro) — esquecer o simétrico é o erro mais comum.

- **Sobreposição:** se dois discos se sobrepõem, a posição permanece zerada (não há "soma" ou restauração).
- **Comparação não estrita:** uma posição é zerada se distância $\leq r$.
- **Ordem de leitura:** os K picos devem ser processados na ordem em que aparecem na entrada, mas o resultado final independe da ordem (operações de zerar são comutativas).

5.13.2.3 🧠 Fundamentação Teórica

Conceito	Papel no filtro *notch*
Pico em $(\Delta v, \Delta u)$	Frequência da interferência periódica detectada visualmente no espectro
Ponto simétrico $(-\Delta v, -\Delta u)$	Toda DFT de sinal real é hermitiana: picos sempre aparecem em pares simétricos ao centro
Raio r	Controla a "largura" da rejeição — r grande remove mais energia ao redor do pico, mas também informação útil

5.13.2.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro L.
- Linha 2: Inteiro C.
- Linhas seguintes: Elementos inteiros da matriz de magnitude (centrada), L linhas.
- Próxima linha: Inteiro K.
- K linhas seguintes: três inteiros Δv, Δu, r (separados por espaço).

Saída:

- Matriz resultante em L linhas e C colunas, separados por espaço.

5.13.2.5 📌 Exemplos

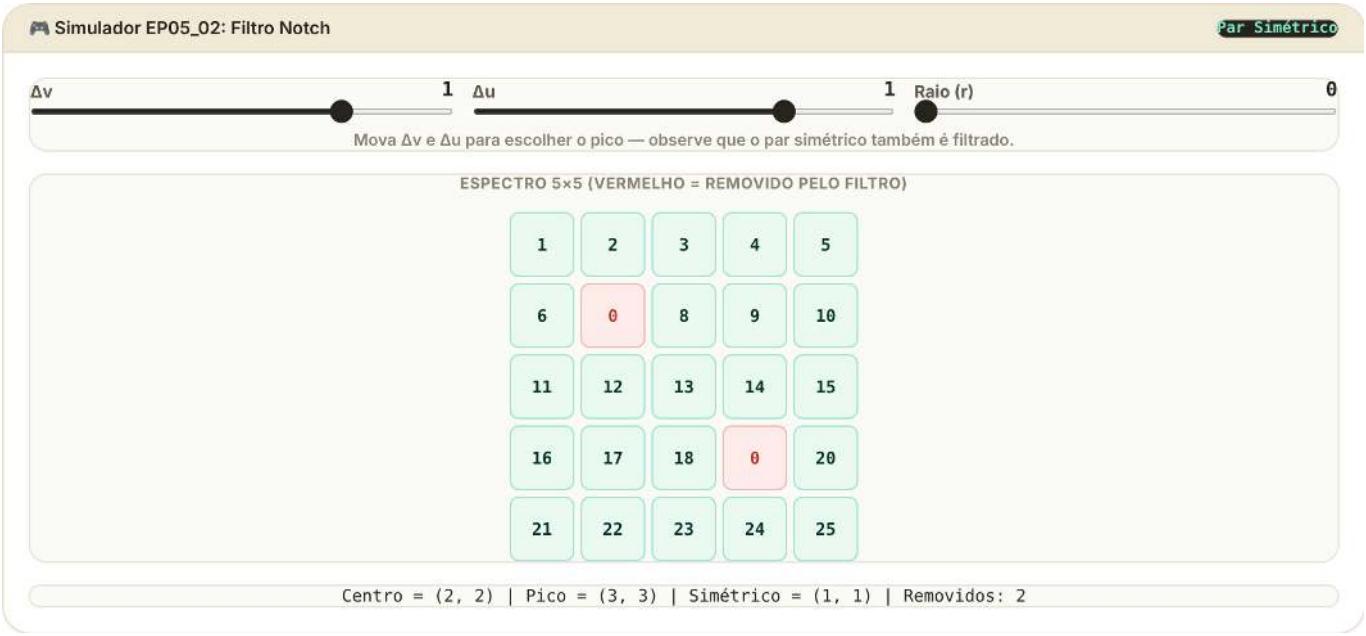
Entrada	Saída	Observação
5 5 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 1 1 1 0	1 2 3 4 5 6 0 8 9 10 11 12 13 14 15 16 17 18 0 20 21 22 23 24 25	Centro $(c_y, c_x) = (2, 2)$. Pico informado $(\Delta v, \Delta u) = (1, 1)$ gera o ponto $(3, 3)$ (valor 19) e seu simétrico $(1, 1)$ (valor 7), ambos zerados com $r = 0$ (apenas os pontos exatos).

```
%%writefile EP05_02.cpp
// sua solução
```

```
Overwriting EP05_02.cpp
```

```
TestSuite("EP05_02.cpp").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap05/sim-ep0502.html

Figura 5.33: Simulador EP05_02: Filtro Notch

5.13.3 EP05_03 🟠 Quantização DCT: a Verdadeira Fonte de Compressão

Um aplicativo de **galeria de fotos** precisa reduzir o tamanho de milhares de imagens antes de fazer *upload* para a nuvem, sem recodificar tudo do zero. O engenheiro responsável já tem os **coeficientes DCT** de cada bloco 4×4 calculados (a etapa cara computacionalmente já foi feita) — falta apenas aplicar a **tabela de quantização**, a etapa que realmente descarta informação e gera compressão. Coeficientes de alta frequência, menos perceptíveis ao olho humano, recebem divisores grandes e tendem a virar **zero**; coeficientes de baixa frequência, mais perceptíveis, recebem divisores pequenos e sobrevivem quase intactos.

Você vai implementar exatamente essa etapa: **quantizar e desquantizar** (dividir, arredondar, multiplicar de volta) — o coração da compressão *lossy* do JPEG.

5.13.3.1 📋 Diretrizes de Implementação

1. **Dimensão do bloco:** Ler o inteiro N (bloco $N \times N$).
2. **Coeficientes:** Ler a matriz C de coeficientes DCT, N linhas com N inteiros cada (podem ser negativos).
3. **Tabela de quantização:** Ler a matriz Q, N linhas com N inteiros positivos cada.
4. **Quantização:** Para cada posição (u, v), calcular o índice quantizado

$$\tilde{C}(u,v) = \text{round}\left(\frac{C(u,v)}{Q(u,v)}\right)$$

   usando arredondamento padrão para o inteiro mais próximo (valores intermediários `.5` nunca ocorrem nos casos de teste).
5. **Desquantização (reconstrução):** Calcular

$$C'(u,v) = \tilde{C}(u,v) \times Q(u,v)$$

6. **Saída:** Exibir a matriz reconstruída C', $N \times N$, inteiros.

5.13.3.2 📌 Restrições Computacionais

- ***Round-trip* completo:** a saída é o coeficiente **reconstruído** ($\tilde{C} \times Q$), não o índice quantizado isolado.
- **Divisão em ponto flutuante:** a divisão $C(u,v)/Q(u,v)$ deve ser feita em ponto flutuante antes do arredondamento — divisão inteira truncada produzirá resultado incorreto.

- **Sinal preservado:** coeficientes negativos mantêm o sinal após quantização e reconstrução.
- $Q(u, v) > 0$ **sempre:** não há necessidade de tratar divisão por zero.

5.13.3.3 🧠 Fundamentação Teórica

Coeficiente	Frequência	Valor típico de Q	Efeito da quantização
$C(0,0)$	DC (média do bloco)	Pequeno	Quase sempre sobrevive — domina a energia
$C(u,v)$ baixo $u+v$	Baixa frequência	Pequeno/médio	Parcialmente preservado
$C(u,v)$ alto $u+v$	Alta frequência	Grande	Frequentemente vira zero — fonte da compressão

5.13.3.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro N.
- N linhas seguintes: matriz C (coeficientes DCT, inteiros, podem ser negativos).
- N linhas seguintes: matriz Q (tabela de quantização, inteiros positivos).

Saída:

- Matriz reconstruída C', $N \times N$, inteiros separados por espaço.

5.13.3.5 📌 Exemplos

Entrada	Saída	Observação
4 50 10 -5 0 8 -3 2 1 0 1 0 0 2 0 0 -1 2 5 7 8 4 7 8 11 6 8 11 12 9 11 12 14	50 10 -7 0 8 0 0 0 0 0 0 0 0 0 0 0	$C(0,0) = 50/2 = 25 \to 25 \times 2 = 50$ (preservado). $C(0,2) = -5/7 \approx -0.71 \to -1 \to -1 \times 7 = -7$. Já $C(1,1) = -3/7 \approx -0.43 \to 0$: zerado pela quantização — a maior parte do bloco vira zero, ilustrando a compactação de energia no canto superior esquerdo.

```
%%writefile EP05_03.cpp
// sua solução
```

```
Overwriting EP05_03.cpp
```

```
TestSuite("EP05_03.cpp").run()
```

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

5.13.4 EP05_04 🔴 Implementando a DFT 2D a Partir da Definição

Um laboratório de pesquisa em **astronomia computacional** recebeu, de uma missão antiga, um pequeno sensor experimental cujos dados brutos não podem ser processados por bibliotecas modernas de FFT — o ambiente de validação é isolado e só permite operações aritméticas básicas. A equipe precisa **reimplementar a Transformada de Fourier Discreta 2D a partir da própria definição matemática**, célula por célula, para depois comparar bit a bit com `np.fft.fft2` em outro ambiente.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap05/sim-ep0503.html

Figura 5.34: Simulador EP05_03: Quantização DCT (*round-trip*)

Este é o exercício mais conceitual da lista: não há atalhos. Você vai implementar o duplo somatório da Equação 5.1 diretamente, evidenciando *por que* a FFT existe — e o custo computacional que ela evita.

5.13.4.1 📋 Diretrizes de Implementação

1. **Dimensões:** Ler os inteiros M (linhas) e N (colunas) da imagem $f(x, y)$.
2. **Dados:** Ler os valores inteiros de $f(x, y)$, linha a linha.
3. **DFT 2D:** Para cada par de frequências (u, v) com $u = 0, \ldots, M-1$ e $v = 0, \ldots, N-1$, calcular

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y)\, e^{-j2\pi\left(\frac{ux}{M} + \frac{vy}{N}\right)}$$

   usando a identidade de Euler $e^{-j\theta} = \cos(\theta) - j\sin(\theta)$ para separar parte real e imaginária — **não utilize nenhuma função de FFT pronta**.
4. **Magnitude:** Calcular $|F(u, v)| = \sqrt{\mathrm{Re}(F)^2 + \mathrm{Im}(F)^2}$ e arredondar para o inteiro mais próximo.
5. **Saída:** Exibir a matriz de magnitudes arredondadas, $M \times N$, na mesma ordem (sem `fftshift` — o DC permanece em $(0, 0)$).

5.13.4.2 📌 Restrições Computacionais

- **Proibido usar bibliotecas de FFT:** a implementação deve calcular os somatórios duplos explicitamente (laços aninhados), mesmo que mais lenta.
- **Sem `fftshift`:** a saída mantém a convenção crua da DFT, com o componente DC em $F(0, 0)$ (canto superior esquerdo).
- **Arredondamento:** a magnitude final deve ser arredondada para o inteiro mais próximo; nos casos de teste não há ambiguidade `.5`.
- **Precisão:** pequenos erros de ponto flutuante (ordem de 10^{-6}) antes do arredondamento são esperados e não afetam o resultado inteiro final.

5.13.4.3 🧠 Fundamentação Teórica

Elemento	Significado
$F(0, 0)$	Componente DC — soma de todos os pixels, $F(0, 0) = \sum f(x, y)$
Parte real $\mathrm{Re}(F)$	Projeção do sinal sobre cossenos

Elemento	Significado
Parte imaginária $\mathrm{Im}(F)$	Projeção do sinal sobre senos
Complexidade desta implementação	$\mathcal{O}((MN)^2)$ — por isso a FFT, com $\mathcal{O}(MN\log(MN))$, é indispensável em imagens reais

5.13.4.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro M.
- Linha 2: Inteiro N.
- Linhas seguintes: Elementos inteiros de $f(x, y)$, M linhas.

Saída:

- Matriz de magnitudes $|F(u, v)|$ arredondadas, $M \times N$, separadas por espaço.

5.13.4.5 📌 Exemplos

Entrada	Saída	Observação
2 2 1 2 3 4	10 2 4 0	$F(0,0) = 1+2+3+4 = 10$ (DC = soma total). $F(0,1) = (1-2)+(3-4) = -2 \to \lvert F\rvert = 2$. $F(1,0) = (1+2)-(3+4) = -4 \to \lvert F\rvert = 4$. $F(1,1) = (1-2)-(3-4) = 0$.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap05/sim-ep0504.html

Figura 5.35: Simulador EP05_04: DFT 2D manual

```
%%writefile EP05_04.cpp
// sua solução
```

```
Overwriting EP05_04.cpp
```

```
TestSuite("EP05_04.cpp").run()
```

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

5.13.5 EP05_05 🏆 *Pipeline* JPEG Completo: DCT, Quantização e Reconstrução

Você foi contratado para criar, do zero, um **codec JPEG didático** em ambiente embarcado, sem qualquer biblioteca de imagem disponível — apenas operações matemáticas básicas. O cliente quer entender exatamente onde a qualidade é perdida e onde ela é recuperada, bloco por bloco. Este é o desafio final do capítulo: integrar **tudo** o que foi estudado — a DCT-II ortonormal, a quantização perceptual e a reconstrução via IDCT — em um único *pipeline* de ponta a ponta, processando um bloco $N \times N$ do início ao fim, exatamente como o padrão JPEG faz internamente, 8×8 pixels de cada vez.

5.13.5.1 📋 Diretrizes de Implementação

1. **Dimensão do bloco:** Ler o inteiro N.
2. **Bloco original:** Ler a matriz de pixels $f(x, y)$, N linhas com N inteiros em $[0, 255]$.
3. **Tabela de quantização:** Ler a matriz Q, $N \times N$ inteiros positivos.
4. **Centralização:** Subtrair 128 de cada pixel: $g(x, y) = f(x, y) - 128$.
5. **DCT-II 2D ortonormal:** Calcular

$$C(u, v) = \alpha(u)\,\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} g(x, y)\, \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right]$$

   com $\alpha(0) = \sqrt{1/N}$ e $\alpha(k) = \sqrt{2/N}$ para $k > 0$.
6. **Quantização:** $\tilde{C}(u, v) = \text{round}(C(u, v)/Q(u, v))$.
7. **Desquantização:** $C'(u, v) = \tilde{C}(u, v) \times Q(u, v)$.
8. **IDCT-II 2D (inversa ortonormal):** Calcular $g'(x, y)$ a partir de $C'(u, v)$ usando a transformada inversa correspondente (mesma base, somatório sobre u, v).
9. **Reversão da centralização e arredondamento:** $f'(x, y) = \text{round}(g'(x, y) + 128)$, restrito ao intervalo $[0, 255]$ (*clipping*).
10. **Saída:** Exibir o bloco reconstruído f', $N \times N$, inteiros.

5.13.5.2 📌 Restrições Computacionais

- ***Pipeline* completo obrigatório:** todas as seis etapas (centralizar, DCT, quantizar, desquantizar, IDCT, reverter) devem ser implementadas — pular a quantização não passa nos testes, pois o resultado seria idêntico ao original.
- ***Clipping*:** valores reconstruídos fora de $[0, 255]$ devem ser truncados (0 se negativo, 255 se maior que 255).
- **Arredondamento:** tanto na quantização quanto na reconstrução final dos pixels, use arredondamento padrão; os casos de teste evitam ambiguidade `.5`.
- **Base ortonormal:** a normalização $\alpha(u)$ e $\alpha(v)$ deve ser aplicada exatamente como especificado — sem ela, a IDCT não reconstrói corretamente.

5.13.5.3 🧠 Fundamentação Teórica

Etapa	Análoga no padrão JPEG real	Onde a qualidade é perdida
Centralização	Mesma — DCT assume sinal centrado em zero	Nenhuma perda
DCT-II	Etapa 3–4 do *pipeline* (Tabela 5.7)	Nenhuma perda (transformação exata e reversível)
Quantização	Etapa 5 — divisão por $Q(u, v)$	**Principal fonte de perda** — coeficientes de alta frequência viram zero
IDCT	Reconstrução final	Reconstrói exatamente os coeficientes *quantizados*, não os originais

5.13.5.4 📦 Especificação de Entrada e Saída (VPL)

Entrada:

- Linha 1: Inteiro N.
- N linhas seguintes: bloco original $f(x, y)$, inteiros em $[0, 255]$.
- N linhas seguintes: tabela de quantização Q, inteiros positivos.

Saída:

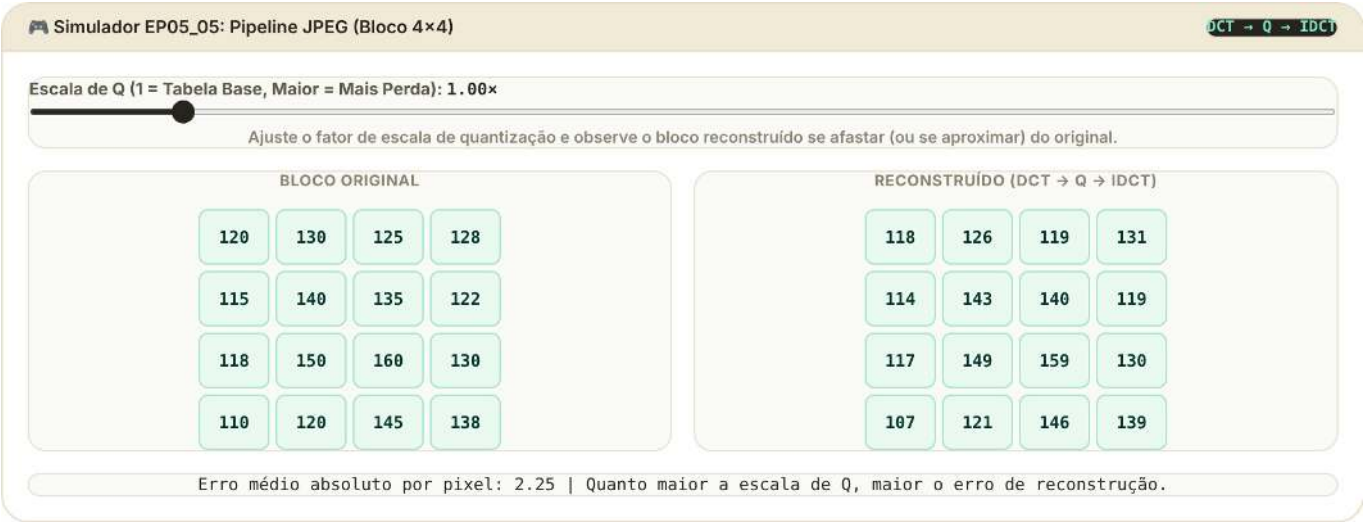
- Bloco reconstruído $f'(x, y)$, $N \times N$, inteiros em $[0, 255]$, separados por espaço.

5.13.5.5 📌 Exemplos

Entrada	Saída	Observação
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	Após DCT, quantização agressiva nas altas frequências (valores grandes de Q no canto inferior direito) e reconstrução via IDCT, o bloco fica **próximo** do original, mas não idêntico — a diferença é o custo da compressão *lossy*.

5.13.5.6 💡 Dica de Depuração

Se o resultado não bater, verifique nesta ordem: (1) os coeficientes DCT brutos (antes da quantização) — eles devem reconstruir o original **exatamente** via IDCT se você pular a etapa 6–7; (2) a tabela $\alpha(u)$ — erro comum é aplicar $\sqrt{2/N}$ também para $u = 0$; (3) o arredondamento da quantização, que deve ocorrer **antes** de multiplicar de volta por Q.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/cpp.pt/cap05/sim-ep0505.html

Figura 5.36: Simulador EP05_05: *Pipeline* JPEG completo em bloco

```
%%writefile EP05_05.cpp
// sua solução
```

```
Overwriting EP05_05.cpp
```

```
TestSuite("EP05_05.cpp").run()
```

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

Referências

BARRERA, Junior *et al.* MMach: a mathematical morphology toolbox for the KHOROS system. **Journal of Electronic Imaging**, v. 7, n. 1, p. 174–210, 1998.

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. *Em*: 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. *Em*: 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.

Apêndice A

Sobre o MCTest

O **MCTest** é um sistema de código aberto para criação e correção automática de exames parametrizados (Zampirolli, 2023). Ele oferece suporte a questões de múltipla escolha, dissertativas e de programação, estas últimas integradas ao VPL do Moodle, descrito no Apêndice B.

A versão do MCTest utilizada neste livro é a **5.4**, disponível em https://github.com/fzampirolli/mctest. A subpasta `book` do repositório contém as versões **5.3**, descrita em (Zampirolli, 2023), e **5.4**, utilizada para gerar as provas apresentadas neste livro e atualmente em produção na UFABC (https://mctest.ufabc.edu.br).

A.1 Instalação do MCTest

A instalação recomendada utiliza o VirtualBox com Ubuntu 22.04. No terminal do Ubuntu, como *root*, execute:

```
wget https://raw.githubusercontent.com/fzampirolli/mctest/master/_setup-all.sh
```

Em seguida, substitua `yourLogin` pelo nome do usuário local e execute:

```
sed -i 's/\/home\/fz\//\/home\/yourLogin\//g' _setup-all.sh
source _setup-all.sh
pip install mysqlclient
```

Após alguns minutos, o MCTest estará configurado. Para executá-lo:

```
source /home/yourLogin/PycharmProjects/runDjango.sh
```

O sistema fica então acessível em `http://127.0.0.1:8000`.

A.2 Criando um exame no MCTest

No MCTest, todo exame é configurado em uma tela de **Exame**, que reúne Turmas, Tópicos (associados a uma disciplina, como PDI-VC) e Questões — cada questão pertence a um único tópico. Além dos atributos de banco de dados, o exame possui parâmetros próprios, como o número de questões sorteadas e o número de variações geradas.

No caso das duas provas descritas neste apêndice, foi definido um conjunto de questões parametrizadas por prova, das quais um subconjunto é sorteado aleatoriamente para cada variação, totalizando **110 variações** distintas — uma por estudante matriculado nas turmas.

O botão **Create-Variations** gera um novo conjunto de variações a cada acionamento. Quando as questões estão associadas a atividades VPL, o professor recebe por e-mail um arquivo `*linker.json` contendo todos os casos de teste das variações geradas. O botão **Create-PDF** gera, para cada turma, um PDF com as provas sorteadas por estudante, além de um arquivo `*students_variations.csv` com nome, e-mail e variação sorteada de cada estudante.

> **!** Atenção
>
> Cada acionamento do botão **Create-Variations** gera um novo conjunto de variações de exames. Após imprimir o PDF, é fundamental **não alterar os atributos do exame**, sob risco de invalidar a correção automática das provas já impressas ou aplicadas.

A.3 Criando uma questão paramétrica

Uma questão paramétrica no MCTest combina três elementos:

1. **Descrição** — o enunciado da questão, escrito em LaTeX, contendo variáveis entre `[[code:variavel]]`, que são substituídas pelo valor sorteado para cada estudante;
2. **Bloco de definição** (`[[def: ... ]]`) — um trecho de código Python, embutido na própria questão, responsável por sortear os parâmetros, calcular a solução de referência e montar os casos de teste;
3. **Casos de teste para o Moodle** (bloco `moodle_cases`) — dicionário serializado em JSON contendo as entradas e saídas esperadas, consumido pela atividade VPL.

Um exemplo simplificado de definição de questão (adaptado da geração de um tabuleiro de xadrez `h × w`) é:

```
[[code:texto]]

\vspace{2mm}\noindent\textbf{Exemplo de Entrada:}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_inp]]
\end{verbatim}
\vspace{-2mm}\noindent\textbf{Exemplo de Saída:}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_out]]
\end{verbatim}

\begin{comment}
[[code:moodle_cases]]
\end{comment}

[[def:
import json
import numpy as np
from topic.morph import mm # TEM QUE INCLUIR "topic." no MCTest

def chess(h, w):
    m = np.zeros((h, w), dtype='int')
    for i in range(h):
        for j in range(w):
            if (i + j) % 2:
                m[i][j] = 1
    return m

# PARÂMETROS USADOS NA DESCRIÇÃO DA QUESTÃO
height = int(np.random.randint(5, 15))

# ENUNCIADO COMPLETO, COM O VALOR DE height JÁ INTERPOLADO
texto = (
    r"\vspace{2mm}\noindent\textbf{Descrição:}\vspace{-2mm} "
    r"Escreva um programa que leia um número inteiro \texttt{W}, representando a "
    r"largura (número de colunas) de um tabuleiro, e imprima esse tabuleiro no "
    r"formato de um tabuleiro de xadrez, usando os dígitos \texttt{0} e \texttt{1}. "
    r"O tabuleiro impresso terá sempre altura (número de linhas) igual a "
    f"\\textbf{{{height}}} "
    r"e largura igual ao valor \texttt{W} lido da entrada. "
```

```
    r"Considerando a posição $(i, j)$ do tabuleiro (indexada a partir de 0, com "
    r"$i$ representando a linha e $j$ a coluna), o valor impresso nessa posição "
    r"deve ser \texttt{1} se $i + j$ for ímpar, e \texttt{0} caso contrário. Cada "
    r"linha do tabuleiro deve ser impressa em uma linha separada, com os valores "
    r"de cada coluna separados por um espaço."
)

inp_list, out_list, test_cases = [], [], 4
for i in range(test_cases):
    width = int(np.random.randint(500, 1500) / 100)
    inp = str(width) + '\n'
    out = mm.drawImg(chess(height, width)) + '\n'
    inp_list.append(inp)
    out_list.append(out)

cases = {}
cases['skills']      = ["matrizes", "laços de repetição", "formatação de saída"]
cases['description'] = [{"text": latex_to_text(texto)}]
cases['input']       = inp_list
cases['output']      = out_list

moodle_cases = json.dumps(cases)

caso0_inp = cases['input'][0]
caso0_out = cases['output'][0]
]]
```

O valor de `height` é sorteado uma única vez por variação (para aparecer no enunciado), enquanto `width` é sorteado a cada caso de teste, aumentando a robustez da correção automática.

Vale destacar o papel duplo da variável `texto` nesse mecanismo. Ela é ao mesmo tempo:

- **conteúdo do enunciado impresso**, inserida na descrição LaTeX da questão através da tag `[[code:texto]]`, aparecendo no PDF gerado pelo botão **Create-PDF**; e
- **entrada do dicionário exportado para o Moodle**, através da chave `cases['description']`, que passa a compor o JSON `moodle_cases`. É justamente esse campo que a atividade VPL exibe ao estudante quando ele abre a questão no Moodle para avaliá-la ou submeter uma solução.

Há, porém, uma diferença importante entre esses dois usos: o PDF é composto em LaTeX e, portanto, interpreta normalmente comandos como `\textbf{}`, `\texttt{}` ou `$...$`. Já o VPL do Moodle **não** processa LaTeX — ele espera texto puro. Por isso, ao montar `cases['description']`, `texto` não é inserido diretamente, mas sim passado pela função utilitária `latex_to_text()`, própria do MCTest, que remove (ou converte) os comandos e símbolos LaTeX do enunciado, produzindo uma versão em texto simples equivalente. Assim, `[[code:texto]]` no PDF continua recebendo o `texto` original, com toda a formatação LaTeX, enquanto `cases['description']` recebe `latex_to_text(texto)`, garantindo que o enunciado exibido no Moodle seja legível mesmo sem suporte a LaTeX.

Como `texto` é montada dentro do próprio bloco `[[def: ...]]` — no mesmo escopo em que `height`, `width` e os demais parâmetros são sorteados —, ela é recalculada a cada variação. Isso garante que o enunciado visto pelo estudante no Moodle seja **sempre idêntico** ao que foi impresso na prova em papel, mesmo quando o texto muda de estudante para estudante junto com os valores sorteados. A Seção A.4 retoma esse ponto em um caso real, no qual o enunciado é significativamente mais longo e descreve, além dos parâmetros numéricos, o próprio cenário visual gerado para cada variação.

A Figura A.1 mostra a questão do tabuleiro de xadrez efetivamente gerada pelo MCTest para uma das variações, com o enunciado já contendo o valor sorteado de `height` e o exemplo de entrada/saída correspondente ao primeiro caso de teste:

Ao acionar **Create-PDF** na tela da questão, uma nova variação é gerada a cada clique, permitindo conferir o enunciado antes de publicá-lo.

1. **Descrição:** Escreva um programa que leia um número inteiro `W`, representando a largura (número de colunas) de um tabuleiro, e imprima esse tabuleiro no formato de um tabuleiro de xadrez, usando os dígitos `0` e `1`. O tabuleiro impresso terá sempre altura (número de linhas) igual a **6** e largura igual ao valor `W` lido da entrada. Considerando a posição (i, j) do tabuleiro (indexada a partir de 0, com i representando a linha e j a coluna), o valor impresso nessa posição deve ser `1` se $i + j$ for ímpar, e `0` caso contrário. Cada linha do tabuleiro deve ser impressa em uma linha separada, com os valores de cada coluna separados por um espaço.

Exemplo de Entrada:

```
9
```

Exemplo de Saída:

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

Figura A.1: Questão do tabuleiro de xadrez gerada pelo MCTest, ilustrando o enunciado parametrizado com o valor de `height` sorteado para a variação e o exemplo de entrada/saída do primeiro caso de teste.

Para as duas provas da disciplina, esse mecanismo foi utilizado de forma mais ampla: cada questão paramétrica define não apenas os valores numéricos, mas também, em alguns casos, pequenas variações estruturais no enunciado (por exemplo, qual direção cardinal — Norte, Sul, Leste, Oeste — deve ser avaliada), dificultando a busca por soluções prontas na internet ou em ferramentas de IA generativa.

A.4 Exemplo real: uma questão do Simulado 4

Para ilustrar o mecanismo descrito na seção anterior com um caso concreto, apresenta-se a seguir uma questão efetivamente aplicada no **Simulado 4** (tópico *im4-Operadores Morfológicos*, dificuldade 1, taxonomia de Bloom "remember"), do tipo questão de programação parametrizada com integração Moodle+VPL.

A questão pede ao estudante que escreva um programa capaz de:

- ler uma imagem binária, corrompida por ruído sal e pimenta, contendo ao menos dois objetos geométricos isolados;
- aplicar filtragem morfológica (abertura seguida de fechamento) para eliminar o ruído;
- extrair, a partir da imagem filtrada, as medidas geométricas de cada objeto (área, perímetro, centro, bounding box, circularidade, solidez e número de vértices), usando a função auxiliar `mm.measure`;
- ordenar os objetos por posição (coordenada X do bounding box, com Y como critério de desempate) e reatribuir os identificadores sequencialmente;
- imprimir a matriz limpa e a tabela de medidas no formato esperado pelo corretor automático.

No bloco `[[def: ... ]]` correspondente, um gerador de cena (`gerarCena`) sorteia aleatoriamente a altura e a largura da imagem, o tipo, o tamanho e a posição de cada objeto (quadrado, retângulo, triângulo ou bloco), garantindo que eles não se sobreponham, e em seguida acrescenta ruído sal e pimenta com 3% de probabilidade por pixel. Dez casos de teste distintos são gerados dessa forma para cada variação da prova, e o par entrada/saída do primeiro caso é reaproveitado no próprio enunciado como exemplo para o estudante. A biblioteca de morfologia (`mm`) é importada diretamente de `morph.py`, também disponível como módulo utilitário do próprio MCTest.

Em resumo, o enunciado pede ao estudante um programa que:

1. leia, na primeira linha, dois inteiros H e W (altura e largura da imagem);
2. leia as H linhas seguintes da matriz binária (0s e 1s separados por espaço), usando `mm.readImg(H, W)`;
3. aplique filtragem morfológica para remover o ruído sal e pimenta;
4. imprima a matriz já filtrada, em 0s e 1s separados por espaço;
5. extraia as medidas geométricas dos objetos com `mm.measure(imgLimpa)`;
6. ordene os objetos e imprima a tabela de medidas no formato esperado.

O enunciado ainda traz duas observações importantes para a correção automática: (i) a área calculada

pelo OpenCV (`cv2.contourArea`) corresponde ao polígono contínuo delimitado pelos centros dos pixels de borda, sendo por isso menor do que a simples contagem de pixels 1 (`np.sum`); e (ii) a lista de objetos deve ser ordenada em ordem crescente pela coordenada X do bounding box (`bbox[0]`), usando a coordenada Y (`bbox[1]`) como critério de desempate, com os IDs reatribuídos sequencialmente de 1 a N após a ordenação.

Assim como no exemplo da Seção A.3, o texto do enunciado aqui também não é fixo: ele é montado dentro do próprio bloco `[[def: ... ]]`, na variável Python `texto`, e desempenha o mesmo papel duplo já descrito — alimenta o PDF via `[[code:texto]]` e é exportado, já convertido por `latex_to_text()`, em `cases['description']` dentro de `moodle_cases`, sendo essa a versão exibida ao estudante no VPL do Moodle. A diferença é que, neste caso real, é o **cenário visual** (imagem ruidosa, número e posição dos objetos) que muda de estudante para estudante a cada variação, enquanto a redação de `texto` permanece fixa entre variações, já que apenas os dados numéricos (imagem e casos de teste) são sorteados. O ideal seria que a redação também variasse a cada geração, como no exemplo anterior, em que o valor de `height` era interpolado diretamente no texto da descrição. A estrutura real utilizada (com trechos de sorteio de cenário omitidos por brevidade) é:

```
[[code:texto]]

\vspace{2mm}\noindent\textbf{Exemplo de Entrada:}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_inp]]
\end{verbatim}
\vspace{-2mm}\noindent\textbf{Exemplo de Saída:}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_out]]
\end{verbatim}

\begin{comment}
[[code:moodle_cases]]
\end{comment}

[[def:
import json
import numpy as n
import cv2
from topic.morph import mm # TEM QUE INCLUIR "topic." no MCTest

# ... geração da cena, do ruído e dos 10 casos de teste (inplist/outlist) ...

# ENUNCIADO COMPLETO, COM EXPLICAÇÃO DE ÁREA E DE ORDENAÇÃO
texto = (
    "Uma imagem binária corrompida por ruído sal e pimenta contém no mínimo dois objetos
    geométricos isolados.\n\n"
    "Escreva um programa que:\n"
    "1. Leia dois inteiros ..."
)

cases = {}
cases['skills']      = ["morfologia matematica", "remocao de ruido", "mm.measure",
"tabulacao"]
cases['description'] = [{"text": latex_to_text(texto)}]
cases['input']       = np.array(inplist).tolist()
cases['output']      = np.array(outlist).tolist()

moodle_cases = json.dumps(cases)
caso0_inp = cases['input'][0]
caso0_out = cases['output'][0]
]]
```

Abaixo, a fotografia da questão efetivamente gerada e impressa para uma das 110 variações do Simulado

4, mostrando o enunciado, a imagem de entrada ruidosa e o exemplo de saída esperado (imagem filtrada e tabela de medidas):

1. Uma imagem binária corrompida por ruído sal e pimenta contém no mínimo dois objetos geométricos isolados.

Escreva um programa que: 1. Leia dois inteiros na primeira linha representando a altura (H) e largura (W) da imagem. 2. Leia as H linhas seguintes da matriz binária (valores 0 e 1 separados por espaço) usando mm.readImg(H, W). 3. Aplique a filtragem morfológica para eliminar o ruído sal e pimenta. 4. Imprima a matriz limpa diretamente em valores 0 e 1 (0s e 1s separados por espaço). 5. Extraia as medições geométricas dos objetos usando mm.measure(imgLimpa). 6. Ordene a lista de objetos e imprima a tabela formatada das medidas.

Observação sobre a área: A área calculada pelo OpenCV (cv2.contourArea) mede o polígono contínuo delimitado pelos centros dos pixels de borda, resultando em um valor menor do que a simples contagem discreta de pixels 1 (np.sum).

Regra de Ordenação: A lista de objetos deve ser ordenada em ordem crescente pela posição X do retângulo delimitador (bbox[0]), utilizando a posição Y (bbox[1]) como critério de desempate caso dois objetos comecem na mesma coluna (medidas.sort(key=lambda m: (m['bbox'][0], m['bbox'][1]))). Em seguida, reatribua os IDs sequencialmente de 1 a N.

Exemplo de Entrada:

```
12 43
0 0 0 0 0 0 0 0 0 0 0 0 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
```

Figura A.2: Questão real gerada pelo MCTest para o Simulado 4 — tópico “Operadores Morfológicos”.

A.5 Considerações finais

Este apêndice resumiu os passos para instalar o MCTest, criar exames e construir questões parametrizadas com integração ao VPL do Moodle. O uso combinado de variáveis sorteadas na descrição (`[[code:...]]`) e de código Python embutido na própria questão (`[[def: ... ]]`) permite gerar centenas de variações de prova a partir de um único modelo, cada uma corrigida automaticamente e de forma consistente — com a variável `texto`, em particular, garantindo que o mesmo enunciado seja exibido tanto no PDF impresso quanto na avaliação da questão pelo estudante no Moodle. O exemplo real do Simulado 4 (Seção A.4) ilustra como esse mecanismo é usado na prática para gerar questões de programação com cenários visuais distintos por estudante. O Apêndice B detalha como essas variações são publicadas e corrigidas no Moodle, e o Apêndice C descreve como restringir o acesso às provas por meio do SEB.

Apêndice B

Sobre o Moodle (VPL)

Este apêndice descreve a configuração de uma atividade **VPL (*Virtual Programming Lab*)** no Moodle, *plugin* utilizado para a submissão e a correção automática dos Exercícios de Programação (EPs), dos simulados quinzenais e das provas descritas neste livro. A versão do Moodle utilizada foi a **4.5**.

B.1 Configurando uma atividade VPL

A criação de um exame no MCTest com integração ao VPL (ver Apêndice A) gera dois arquivos que devem ser renomeados para `linker.json` e `students_variations.csv`. Esses arquivos, juntamente com o `morph.py`, devem ser adicionados aos **Arquivos de Execução** da atividade VPL, além de todos os arquivos contidos em `VPL_modification/V14-AI-feedback.zip`, disponível no repositório do MCTest em https://github.com/fzampirolli/mctest. Todos eles devem ser marcados com a opção **"*Files to keep when running*"**.

Em síntese, a atividade VPL passa a conhecer, para cada estudante, qual variação da questão foi sorteada (por meio de `students_variations.csv`) e qual é o conjunto de casos de teste correspondente (por meio de `linker.json`). Isso permite que a correção seja individualizada, mesmo quando dezenas de estudantes resolvem a mesma atividade com variações distintas.

B.2 Publicando os EPs como atividades VPL

Os Exercícios de Programação (EPs) apresentados ao final de cada capítulo deste livro também podem ser publicados como atividades VPL, seguindo o mesmo mecanismo. O fluxo típico de uso em sala de aula foi:

1. Abertura dos dois *notebooks* Colab do capítulo (teórico e prático), com destaque para os simuladores interativos;
2. Execução dos blocos de código, alterando parâmetros para reforçar a compreensão dos conceitos;
3. Nas aulas em laboratório, submissão das respostas dos EPs diretamente na atividade VPL correspondente, com retorno imediato sobre a aprovação ou reprovação nos casos de teste.

Caso a atividade VPL não seja gerada pelo MCTest, os arquivos `linker.json` e `students_variations.csv` não serão necessários. Entretanto, para utilizar o *feedback* por IA (ver Apêndice D), devem ser adicionados os arquivos contidos em `VPL_modification/V14-EP0_VPL-TEMPLATE.zip`, disponível no mesmo repositório (https://github.com/fzampirolli/mctest), seguindo o procedimento descrito na Seção B.1.

> **Reaproveitamento dos casos de teste**
>
> Os arquivos `.cases` utilizados na validação local (classe `TestSuite`, descrita no corpo deste livro) são compatíveis com o formato esperado pelo VPL, permitindo reaproveitar o mesmo conjunto de testes tanto no desenvolvimento do EP quanto na correção automatizada no Moodle.

Nos simulados quinzenais — aplicados com o SEB (ver Apêndice C) e com bônus de 5% sobre a nota final —, foram utilizadas atividades VPL com EPs semelhantes aos das listas regulares, porém corrigidas sob restrição de acesso à internet, reforçando o caráter avaliativo da atividade.

B.3 Considerações finais

Este apêndice descreveu a configuração de atividades VPL no Moodle para a correção automática de EPs, simulados e provas parametrizadas. A combinação entre MCTest (geração de variações e casos de teste), VPL (submissão e correção) e SEB (restrição de acesso) forma o tripé sobre o qual se apoia a avaliação automatizada e individualizada utilizada ao longo da disciplina.

Apêndice C

Uso do SEB nos simulados quinzenais e avaliações

Além das duas provas com questões parametrizadas (ver Apêndice A), o ***Safe Exam Browser* (SEB)** também foi utilizado para restringir o acesso aos **simulados quinzenais** — atividades VPL (*Virtual Programming Lab*) com Exercícios de Programação (EPs) semelhantes aos das listas regulares, valendo um bônus de até 5% sobre a nota final —, bem como às demais avaliações práticas.

O SEB garante um ambiente seguro de avaliação ao bloquear o acesso a recursos externos não autorizados nas máquinas do laboratório. Para a implementação adotada neste trabalho, utilizou-se o sistema operacional **Windows 10** e a versão **3.7.1.704** do SEB, disponível em https://safeexambrowser.org/, que deve estar previamente instalada em todas as estações do laboratório.

A seguir, descreve-se o procedimento para configurar o arquivo de ambiente do SEB e vinculá-lo à atividade VPL no Moodle.

C.1 Configurando a aplicação no *SEB Tool*

Para gerar o arquivo de configuração `.seb` que será distribuído aos estudantes, abra o utilitário *SEB Tool* e siga os passos descritos a seguir.

1. ***General***
   - No campo ***Start URL***, insira a URL direta da atividade VPL no Moodle.
   - *Nota:* caso seja necessário navegar por mais de uma atividade dentro do mesmo curso, insira a URL principal do curso e, na aba ***Browser***, marque a opção ***Allow navigation back/forward in exam***.
2. ***Network***
   - Inclua as URLs permitidas para navegação. Para restringir o acesso à atividade desejada, utilize uma expressão de filtro no padrão `/mod/vpl/*?id=4624*`, em que o número final corresponde ao identificador da atividade VPL no Moodle. Na Tabela C.1, a última linha contém a URL da página principal do curso (identificador `475`), mas ela também pode ser substituída pela URL de uma seção que contenha várias atividades. Essa configuração é opcional e útil quando há mais de uma atividade VPL. Nesse caso, a primeira linha da tabela deve ser repetida para cada atividade avaliativa que deverá ficar acessível durante o *exame*.
   - **Configurações gerais da aba *Filter***:
     ☒ *Activate URL filtering*
     ☐ *Filter also embedded content*

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

Active	*Regex*	*Expression*	*Action*
[x]	[ ]	`https://moodle.ufabc.edu.br/mod/vpl/*?id=4624*`	*Allow*
[x]	[ ]	`https://moodle.ufabc.edu.br/login/index.php*`	*Allow*
[x]	[ ]	`https://moodle.ufabc.edu.br/course/switchrole.php*`	*Allow*

Active	*Regex*	*Expression*	*Action*
[x]	[ ]	`https://mctest.ufabc.edu.br/compare`	*Allow*
[x]	[ ]	`https://moodle.ufabc.edu.br/enrol/index.php?id=475`	*Allow*

3. ***Security***
   - Caso o laboratório utilize um projetor ou monitor secundário, ajuste o campo ***Maximum allowed number of connected displays*** para um valor maior que 1, evitando que o SEB bloqueie a exibição.
4. ***Config File***
   - Clique em ***Save Settings As...*** e salve o arquivo com a extensão `.seb`.
5. ***Exam***
   - Marque a opção ***Use Browser Exam Key and Configuration Key***.
   - Copie o código gerado no campo ***Browser Exam Key***. *Importante:* copie essa chave somente **após** salvar o arquivo no passo anterior.

C.2 Vinculando a chave do SEB à atividade VPL no Moodle

Após gerar o arquivo e obter a *Browser Exam Key*, acesse o Moodle para aplicar as restrições.

1. Na página da atividade VPL, clique no ícone de engrenagem (**Configurações**).
2. Acesse ***Configuration*** → ***Submission restrictions*** e clique em ***Show more***.
3. Altere a opção ***SEB browser required*** para ***Yes***.
4. Cole a chave copiada no campo ***SEB exam key(s)***.

C.3 Restrição por endereço IP (opcional)

Para garantir que as submissões ocorram exclusivamente a partir dos computadores do laboratório, é possível combinar o SEB com o filtro de rede do Moodle, preenchendo o campo ***Allowed submission from net***.

- **Faixas consecutivas:** utilize um hífen para definir o intervalo. Exemplo: `111.11.11.1-62` (permite os IPs de `111.11.11.1` a `111.11.11.62`).
- **IPs não consecutivos:** informe os endereços individuais separados por vírgulas.

C.4 Considerações finais

A integração do SEB com o VPL no Moodle cria um fluxo simples para o estudante: basta dar um duplo clique no arquivo `.seb` disponibilizado pelo docente para que o navegador seguro seja aberto diretamente na atividade configurada.

A combinação da *Browser Exam Key* com a restrição por faixa de endereços IP das máquinas do laboratório fornece uma solução robusta para avaliações de programação *online*, dificultando acessos indevidos e contribuindo para a integridade do processo de avaliação.

Apêndice D

Geração de material didático com o *Gemini Notebook*

Este apêndice descreve o uso do ***Gemini Notebook*** **(*NotebookLM*)** como ferramenta de apoio à preparação do material didático da disciplina, complementando o uso de IA na revisão de textos e códigos, mencionado no prefácio deste livro.

D.1 Vídeos conceituais e de resolução de EPs

Para cada capítulo, foi criado um projeto no *Gemini Notebook* tendo como fontes o PDF completo do livro e o arquivo `morph.py`. A partir dessas fontes, o *Gemini Notebook* gerou automaticamente um vídeo curto (em média, 7 minutos), utilizado na abertura das aulas. Os capítulos alternaram dois tipos de vídeo:

- **Vídeo conceitual**, apresentando os fundamentos teóricos do capítulo, geralmente exibido na primeira aula dedicada ao tema;
- **Vídeo de resolução de EPs**, apresentando uma estratégia de solução para os Exercícios de Programação, exibido na aula seguinte, quando a maior parte dos EPs era resolvida em conjunto com a turma.

i Papel do vídeo na aula

O vídeo gerado pelo *Gemini Notebook* não substitui a explicação do professor. Ele funciona como uma breve introdução e motivação ao tema, contextualizando o estudante antes da apresentação dos *slides* e da abertura dos *notebooks Colab*.

D.2 *Slides* de apoio

Complementando os vídeos, o *Gemini Notebook* também gerou, a partir das mesmas fontes (o PDF do livro e o arquivo `morph.py`), um conjunto de aproximadamente 12 *slides* por capítulo, abrangendo tanto os conceitos teóricos quanto a parte prática. A geração utilizou um *prompt* específico para cada capítulo, delimitando o escopo do conteúdo a ser sintetizado e evitando tanto a antecipação de assuntos de capítulos posteriores quanto a reprodução de trechos extensos do livro sem adaptação didática.

Após a apresentação dos *slides*, a aula prosseguia com a abertura dos dois *notebooks Colab* do capítulo (teórico e prático), enfatizando os simuladores interativos e a execução dos blocos de código com alteração de parâmetros.

D.3 Material principal e considerações finais

A geração de vídeos e *slides* com o *Gemini Notebook*, a partir do próprio livro e da biblioteca `morph.py`, permitiu padronizar a abertura das aulas e reduzir o tempo de preparação do material didático, sem prescindir da curadoria do professor na elaboração dos *prompts* e na condução das atividades em sala de aula.

Os vídeos e *slides* produzidos pelo *Gemini Notebook* têm caráter **motivacional** e servem como ponto de partida para a discussão dos conceitos apresentados em cada capítulo. Como qualquer conteúdo gerado por IA, eles podem conter imprecisões ou erros ocasionais. Em alguns casos, essas imprecisões são exploradas intencionalmente durante a aula para verificar se os estudantes estão acompanhando criticamente a apresentação e conseguem identificar inconsistências conceituais.

O conteúdo oficial da disciplina, entretanto, é o material produzido pelo método descrito no prefácio deste livro, disponibilizado em três formatos complementares: **HTML**, para navegação com simuladores interativos; **PDF**, para leitura e impressão; e ***notebooks Jupyter*** (`.ipynb`), para execução e experimentação dos códigos. Todo o material gerado pelo *Gemini Notebook* deve ser entendido como um recurso complementar a esse conteúdo principal.

Assim como nos demais usos de IA generativa descritos neste livro, a responsabilidade pela qualidade técnica, pela correção conceitual e pela adequação pedagógica do material permanece com o professor.

Apêndice E

Uso de IA na Avaliação de Estudantes: o Projeto vpl-ai-feedback

Este apêndice descreve o **vpl-ai-feedback**, sistema de código aberto desenvolvido para apoiar a correção de submissões de código enviadas ao VPL (Moodle) por meio de *feedback* gerado por Inteligência Artificial (IA). Diferentemente de um corretor automático tradicional — que apenas compara saídas com casos de teste —, o **vpl-ai-feedback** envia o código do estudante, junto com a rubrica da questão, a um modelo de linguagem (LLM), que devolve uma análise qualitativa por critério, no estilo de um ***feedback* socrático**: o estudante recebe comentários que apontam o que foi feito corretamente, o que pode ser melhorado e por quê, tanto em atividades **formativas** (exercícios de prática) quanto em atividades **avaliativas** (simulados) do VPL.

Esse mecanismo de *feedback* socrático em atividades VPL foi proposto por Zampirolli et al. (2026). O estudo descreve a integração de IA ao processo de avaliação de exercícios de programação parametrizados no VPL/Moodle, utilizando um conjunto de sete LLMs adotados para analisar o código dos estudantes e gerar *feedback* automatizado a cada solicitação de correção, com resultados preliminares indicando percepção positiva dos estudantes quanto à geração de ideias, clareza das explicações e autonomia de aprendizagem.

O restante deste apêndice detalha a implementação prática dessas ideias no projeto **vpl-ai-feedback** — cujo código está disponível em https://github.com/fzampirolli/vpl-ai-feedback — e descreve especificamente seu uso nas três turmas de PDI-VC entre maio e agosto de 2026.

> ⚠ Atenção: IA não substitui o julgamento docente
>
> O uso de IA para **avaliar** estudantes **não é recomendável como fonte única de nota**, pois modelos de linguagem podem **alucinar** — isto é, produzir avaliações, critérios ou justificativas plausíveis, porém incorretas, mesmo diante de um código correto ou incorreto. Um LLM pode atribuir nota alta a um código com erro sutil, ou nota baixa a um código correto mas escrito de forma incomum, simplesmente porque a explicação gerada "parece" coerente. **A IA deve ser usada apenas como um complemento à avaliação**, nunca como substituta do professor. O Apêndice A e o Apêndice B descrevem os mecanismos de correção objetiva (casos de teste) que continuam sendo a base da nota oficial; este apêndice trata exclusivamente do uso da IA como camada adicional de *feedback* qualitativo.

E.1 Contexto de uso

O **vpl-ai-feedback** foi utilizado em três turmas da disciplina **PDI-VC**, ministradas entre maio e agosto de 2026, totalizando aproximadamente uma centena de estudantes. O *feedback* por IA foi empregado de três formas complementares:

1. ***Feedback* contínuo em atividades formativas** — exercícios de prática enviados ao VPL ao longo do curso, nos quais o estudante podia reenviar o código quantas vezes desejasse. A cada submissão, recebia um *feedback* qualitativo gerado por IA, além da correção objetiva realizada pelo VPL, descrita em Zampirolli et al. (2026).
2. ***Feedback* complementar em simulados (atividades avaliativas bônus)** — quatro simulados realizados ao longo do quadrimestre, previstos no plano de ensino como atividades bônus, cada um

valendo até 5% da nota final. Aplicados aproximadamente 30 minutos antes do término das aulas práticas quinzenais em laboratório, os simulados utilizavam o **vpl-ai-feedback** para gerar uma rubrica individual de avaliação, enviada por e-mail juntamente com um resumo comparativo entre a nota atribuída pelo Moodle/VPL e a avaliação produzida pela IA.

3. ***Feedback* complementar nas Provas 1 e 2 (avaliações oficiais)** — as duas provas formais do quadrimestre, cada uma valendo uma parcela maior da nota final da disciplina, também passaram a utilizar o **vpl-ai-feedback** após sua aplicação. Diferentemente dos simulados (bônus), aqui a nota oficial já é definida pela correção objetiva do VPL e/ou avaliação manual do professor antes da divulgação; a rubrica gerada pela IA é enviada ao estudante apenas posteriormente, como material de apoio para revisão do próprio desempenho — reforçando, também nas avaliações de maior peso, a separação entre *nota oficial* e *feedback* complementar detalhada na Seção D.7.

Uma pesquisa de opinião, respondida por 102 estudantes antes da adoção do sistema, indicou elevada aceitação da proposta. Apenas 2 estudantes atribuíram nota 1 (rejeição) e 7 atribuíram nota 2 ao interesse em receber *feedback* automático de código por IA. Outros 19 declararam-se indiferentes (nota 3), enquanto a maioria — 38 e 37 estudantes — atribuiu notas 4 e 5 (alta aprovação), respectivamente, totalizando os 102 respondentes. Esse resultado motivou a adoção do sistema como complemento ao processo formativo, e não como substituto da correção humana.

! A nota oficial nunca foi a nota da IA

É fundamental destacar que, conforme definido no plano de ensino, **a nota utilizada tanto para o cálculo do bônus de cada simulado quanto para a nota oficial das Provas 1 e 2 foi sempre a nota atribuída pelo VPL** (baseada nos casos de teste) e/ou pela avaliação manual do professor — e **não** a nota sugerida pela IA. A rubrica gerada pelo **vpl-ai-feedback** foi enviada ao estudante apenas como material de apoio ao aprendizado — nunca como critério de atribuição de nota, seja em atividades bônus ou em avaliações oficiais. Essa separação entre *nota oficial* (VPL/professor) e *feedback complementar* (IA) é o princípio central deste apêndice e foi retomada na Seção D.8.

E.2 Arquitetura do projeto

O **vpl-ai-feedback** é um *script* Python assíncrono organizado da seguinte forma:

```
.
  config.yaml              ← configuração (credenciais, provedor, pesos) - NUNCA versionado
  config.yaml.example      ← template público de configuração
  main.py                  ← script principal (async), orquestra todo o processo
  run.sh                   ← wrapper de execução (bash run.sh config.yaml)
  gerar_relatorio.py       ← converte o consolidado *_ALL.txt em CSV
  enviar_email.py          ← envia as rubricas por e-mail a cada estudante
  providers/                ← clientes das APIs de LLM
      __init__.py          ← fábrica de clientes (Factory)
      base.py               ← lógica de retry, backoff e fallback entre modelos
      groq.py
      deepseek.py
      gemini.py
  core/                     ← lógica de negócio
      grader.py            ← identifica arquivos por questão, monta o prompt, chama a LLM
      utils.py
  <pasta_da_turma>/         ← submissões baixadas do Moodle VPL
      Nome estudante - login/
          <timestamp>/               ← última submissão do estudante
              Q1.py                  ← padrão para provas com múltiplas questões
              Q2.py
              rubrica.txt            ← gerado pela LLM (somente se avaliação válida)
```

```
        <timestamp>.ceg/
            execution.txt            ← nota/objetiva atribuída pelo VPL
    ...
```

O sistema aceita três provedores de LLM — **Groq**, **DeepSeek** e **Gemini** — configuráveis em `config.yaml` por meio da chave `llm.provider`. Cada provedor pode ter **múltiplos modelos** cadastrados; a cada chamada, a lista é **embaralhada**, distribuindo a carga entre os estudantes processados em paralelo e reduzindo a chance de erros de limite de requisições (HTTP 429). Quando um modelo falha, o sistema tenta até três vezes antes de passar ao próximo da lista, respeitando o tempo de espera sugerido pelo provedor. Erros de autenticação (401) ou de saldo insuficiente (402) interrompem imediatamente a tentativa naquele provedor.

Um aspecto importante do projeto diz respeito à privacidade dos dados. **Nome, e-mail, login, RA e CPF do estudante nunca são enviados à API do LLM**. São transmitidos para processamento apenas: o nome do arquivo contendo o código-fonte; o próprio código (com os comentários removidos); o *prompt* da rubrica — que não contém dados pessoais —; e, quando disponíveis no `execution.txt` gerado pelo VPL, o enunciado oficial da questão sorteada para aquele estudante e o resultado bruto dos casos de teste executados (entrada, saída produzida pelo código e saída esperada). Esses dois últimos itens, embora extraídos de um arquivo específico de cada estudante, também não contêm identificação pessoal — apenas o texto da questão e os valores de entrada/saída dos testes automáticos —, e são usados como evidência objetiva para reduzir o risco de a IA identificar incorretamente o tipo de questão ou avaliar a lógica do código apenas por leitura estática (Seção D.3).

Ainda assim, é importante reconhecer que os três provedores atualmente suportados — Groq, DeepSeek e Gemini — são serviços comerciais operados por empresas estrangeiras, cujos modelos são executados em infraestrutura fora do país e sujeitos a políticas de uso, disponibilidade e custo que fogem ao controle da instituição de ensino. Essa dependência de provedores externos traz riscos que vão além da correção de código pontual: mudanças de preço, descontinuação de modelos, indisponibilidade temporária do serviço, alterações unilaterais nos termos de uso e, em casos mais sensíveis, restrições geopolíticas de acesso podem comprometer a continuidade de um sistema de avaliação que passe a depender estruturalmente dessas APIs. Por isso, é estratégico que Instituições de Ensino Superior (IES) como a UFABC invistam no desenvolvimento e na hospedagem de **provedores próprios de IA** — por exemplo, modelos abertos (*open-weight*) executados em infraestrutura local ou em nuvem soberana —, reduzindo a dependência de serviços externos, especialmente de outros países, e ampliando o controle institucional sobre custos, disponibilidade, privacidade dos dados de estudantes e continuidade pedagógica de longo prazo. O desenho do **vpl-ai-feedback** já favorece esse caminho: a arquitetura de `providers/` foi construída como uma fábrica (*factory*) extensível, na qual um novo provedor — inclusive um modelo hospedado localmente pela própria instituição, com interface compatível — pode ser adicionado com baixo esforço de integração.

E.3 O *prompt* universal e a rubrica em duas etapas

Cada tipo de prova é descrito em um arquivo de *prompt* (por exemplo, `Simulado4.txt`), referenciado em `grading.prompt_file` no `config.yaml`. Esse arquivo instrui a LLM a realizar a avaliação em **duas etapas**:

1. **Identificação da questão** — a LLM determina o tipo específico de operação sorteado para aquele estudante (útil quando há questões paramétricas sorteadas e embaralhadas por estudante, como descrito no Apêndice A);
2. **Aplicação da rubrica correspondente** — a LLM avalia o código segundo critérios pontuados, cada um com uma faixa máxima de pontos, retornando ao final uma nota no formato `NOTA FINAL: X/PESO`.

Em provas parametrizadas, o arquivo de *prompt* costuma conter **várias rubricas paralelas**, uma para cada tipo de questão possível, delimitadas por marcadores como `[START_RUBRICA_TIPO_A]` / `[END_RUBRICA_TIPO_A]`. A LLM identifica primeiro qual tipo foi sorteado para aquele estudante e aplica **apenas** a rubrica correspondente, ignorando as demais — o que evita que critérios de um tipo diferente contaminem a nota.

Quando a prova possui mais de uma questão por atividade VPL, os arquivos de código devem seguir o

padrão `Q1.*`, `Q2.*` etc. — um arquivo por questão, com extensão livre conforme a linguagem escolhida pelo estudante —, e cada questão tem peso configurável individualmente em `grading.weights` no `config.yaml`. Quando a prova tem uma única questão e não foi gerada pelo MCTest, o sistema aceita qualquer nome de arquivo com extensão suportada, desde que seja o único arquivo de código presente na pasta de submissão.

No caso do Simulado 4 (tópico *Operadores Morfológicos*), por exemplo, um dos tipos possíveis definia três critérios, com peso total de 100 pontos para aquela questão:

Critério	Descrição	Pontuação máxima
1 — Entrada de dados	Leitura de `H`, `W` e da matriz binária (`mm.readImg`)	25 pts
2 — Processamento morfológico	Abertura + Fechamento (`mm.sebox()`), `mm.measure`, ordenação por `bbox[0]`/`bbox[1]` e reatribuição de IDs	50 pts
3 — Saída e formatação	Exibição da imagem limpa (`mm.drawImg`) e tabela de medidas formatada	25 pts

O *prompt* exige ainda um **formato de resposta obrigatório** (largura de linha, ordem das seções, marcação `NOTA FINAL:`), o que torna a extração automática da nota e a montagem do relatório consolidado mais confiáveis — embora, como discutido na Seção D.8, isso não elimine o risco de erro semântico na avaliação do conteúdo.

E.4 Duas camadas adicionais de evidência

Para reduzir o risco de a LLM alucinar o tipo de questão ou a corretude do código — o risco central discutido na Seção D.7 —, o **vpl-ai-feedback** passou a fornecer à IA duas fontes de evidência objetiva, extraídas automaticamente do próprio `execution.txt` gerado pelo VPL, além do código-fonte:

- **Enunciado oficial da questão.** Como as questões costumam ser sorteadas e embaralhadas por estudante no MCTest (Apêndice A), o `execution.txt` de cada estudante já traz, no campo "Descrição resumida da questão", o texto exato da questão que caiu para ele. O sistema extrai esse trecho automaticamente e o envia à LLM como evidência **primária** para identificar o tipo/variação da questão — mais confiável do que inferir apenas pela leitura estática do código, especialmente em submissões incompletas ou ambíguas entre dois tipos próximos (por exemplo, erosão "plana" *versus* "com pesos"). Quando o código entregue diverge do que o enunciado pede, o sistema não ignora o enunciado nem "corrige" a leitura silenciosamente: mantém a identificação pelo enunciado, reduz a confiança da avaliação para "Baixa" e explicita a divergência no *feedback* enviado ao estudante.
- **Resultado real da execução no VPL.** Quando disponível, o sistema também envia à LLM os casos de teste efetivamente executados sobre aquele código — entrada, saída produzida e saída esperada —, como evidência objetiva de corretude, a ser usada para confirmar ou refutar a leitura da lógica antes de pontuar os critérios de processamento e de saída, em vez de a IA depender apenas de simular mentalmente o código (o que é especialmente falho em laços com `min`/`max`, deslocamento de índice ou cálculo de limiar de Otsu, onde erros sutis passam despercebidos numa leitura estática).

Além disso, a LLM é instruída a declarar explicitamente seu **nível de confiança** (Alta, Média ou Baixa) na identificação de cada tipo de questão, com justificativa quando a confiança não for Alta. Esse valor é extraído automaticamente e alimenta uma coluna `Revisar_Manualmente` no relatório CSV consolidado (Seção D.4), permitindo ao professor priorizar exatamente os casos em que a própria IA reconhece maior incerteza — em vez de tratar todas as avaliações de uma turma como igualmente confiáveis.

E.5 Fluxo de execução

A execução típica, feita por `bash run.sh config.yaml`, segue os passos:

1. Carrega `config.yaml` e localiza a pasta de submissões (`paths.student_base_dir`);
2. Para cada estudante, identifica a **última submissão** (pasta com o *timestamp* mais recente);
3. Extrai a nota objetiva do Moodle a partir de `*.ceg/execution.txt`;
4. Para cada questão da prova, localiza o arquivo de código (`Q1.*`, `Q2.*`, ou o único arquivo, no caso de prova que não foi gerada pelo MCTest), monta o *prompt* (código + rubrica) e envia a um modelo sorteado da lista configurada;
5. Processa todos os estudantes **em paralelo**, com controle de concorrência;
6. Gera, por estudante, o arquivo `rubrica.txt` — **somente se a IA retornar ao menos uma avaliação válida** para as questões que possuem código; caso contrário, o arquivo não é salvo, forçando nova tentativa na próxima execução;
7. Consolida todos os `rubrica.txt` em um único `*_ALL.txt` e gera um relatório `*_relatorio.csv`, comparando nota do Moodle, nota da IA e a diferença entre ambas, por questão e no total.

Situação	`rubrica.txt` é salvo?
Todas as questões com código avaliadas com sucesso	✅ Sim
IA falhou em ao menos uma questão com código	❌ Não — reprocessa na próxima execução
estudante sem nenhum arquivo de código enviado	❌ Não
Tipo de questão identificado como DESCONHECIDO	❌ Não

E.6 Exemplo ilustrativo: rubrica de uma prova com três questões

i Sobre a origem deste exemplo

O trecho a seguir **não reproduz o código de nenhum estudante específico**. Trata-se de um exemplo reconstruído pelo autor, que reproduz um *padrão* de erro observado repetidamente ao longo das turmas de PDI-VC — a confusão entre uma função morfológica 2D (`mm.ero`/`mm.ero0`) e uma operação 1D sobre sinais — sem citar ou identificar qualquer submissão real. Essa opção evita qualquer risco de violação do direito autoral do estudante sobre seu próprio código-fonte, ainda que anonimizado, já que a titularidade da obra permanece do autor original mesmo quando nome e *login* são removidos.

O trecho a seguir ilustra a saída gerada pelo **vpl-ai-feedback** para um estudante de uma prova com **três questões** (Q1, Q2, Q3), usando o modelo `deepseek-chat`. O resumo comparativo aparece no topo do arquivo, seguido do enunciado oficial de cada questão, do código submetido e da avaliação por critério.

```
RESUMO - IA (deepseek-chat)  x  MOODLE

Peso total : 100 pts

Q1 (IA) :  3.3 / 33 pts
Q2 (IA) :  6.7 / 33 pts
Q3 (IA) : 33.3 / 34 pts

Moodle : (Q1=33 + Q2=0 + Q3=34) = 67 pts
IA     : 43.3 / 100 pts

Diferença (IA - Moodle): -23.7 pts

Q1: tipo identificado = B  | confiança = Baixa (código não
```

```
  corresponde ao enunciado da questão)  REVISAR
Q2: tipo identificado = A  | confiança = Alta (enunciado oficial
  confirma o tipo e a variação)
Q3: tipo identificado = C1 | confiança = Alta (enunciado oficial
  confirma erosão 2D com OpenCV integrada)
```

Q1 — código não corresponde ao enunciado (confiança baixa): o enunciado oficial da questão, extraído do `execution.txt`, pedia uma erosão morfológica **1D** sobre um sinal (leitura de N, M, o sinal e o elemento estruturante B, seguida do mínimo dos vizinhos ativos). O código submetido, no entanto, lê apenas dois inteiros e invoca `mm.ero0` — uma função de erosão **2D** para imagens, não relacionada à operação pedida.

```
Tipo identificado: TIPO B (Erosão 1D - plana)
Confiança: Baixa - código não corresponde ao enunciado da questão (usa
mm.ero0, função de erosão 2D, em vez de implementar erosão 1D plana
sobre um sinal)

Critério 1 - [3.33/10.00 pts]:
- O código lê dois inteiros em uma única linha (que corresponderiam a
  N e M), mas não lê o sinal nem o elemento estruturante B como vetores
  1D. Em vez disso, trata ambos como imagens via mm.readImg, o que não
  corresponde à leitura esperada de um sinal 1D e de B em linhas
  separadas.

Critério 2 - [0.00/16.67 pts]:
- A operação aplicada é mm.ero0(img, element), erosão 2D para imagens,
  não a erosão 1D de sinais pedida no enunciado. A lógica de mínimo dos
  vizinhos ativos com bordas por recorte não está implementada.

Critério 3 - [0.00/6.67 pts]:
- A saída é uma imagem 2D via mm.drawImg, não o sinal 1D resultante em
  uma única linha com valores inteiros separados por espaço, como
  exigido.

NOTA FINAL: 3.33/33.33

Comentários para o estudante: O código apresentado não corresponde ao que o
enunciado pede. A questão sorteada solicita uma erosão morfológica 1D
sobre um sinal unidimensional; o código usa uma função de erosão 2D
para imagens. Releia atentamente o enunciado sorteado antes de escolher
a implementação.
```

Q2 — erro de sintaxe impede a execução (confiança alta, nota baixa por falha real): aqui a IA identificou corretamente o tipo (limiarização com Otsu), com apoio do enunciado oficial, mas o código continha um erro de indentação que impedia a execução — evidenciado tanto pela leitura estática quanto pela ausência de saída válida nos testes do VPL.

```
Tipo identificado: TIPO A (Limiarização - T manual + Otsu)
Confiança: Alta - enunciado oficial confirma o tipo e a variação

Critério 1 - [6.67/10.00 pts]:
- Lê corretamente os três parâmetros L, C e T em linhas separadas e a
  imagem de entrada. Porém, a definição de uma função auxiliar de
  limiarização está indentada incorretamente dentro do escopo global,
  causando um erro de indentação que impede a execução completa do
  programa.

Critério 2 - [0.00/16.67 pts]:
- A binarização com limiar fixo T está corretamente estruturada, mas o
  cálculo do limiar de Otsu depende da função auxiliar malformada, que
```

```
  referencia uma variável fora de escopo. O erro de indentação impede
  qualquer processamento.

Critério 3 - [0.00/6.67 pts]:
- Devido ao erro de indentação, nenhuma saída é produzida.

NOTA FINAL: 6.67/33.33

Comentários para o estudante: A estrutura geral está correta (leitura de L,
C, T e da imagem), mas há um erro de indentação que impede a execução.
Revise a sintaxe Python antes de submeter e teste localmente o código
antes do envio ao VPL.
```

Q3 — divergência de função, mas equivalência funcional confirmada por evidência real (confiança alta, nota justa): o enunciado pedia explicitamente o uso de `mm.ero` (versão integrada ao OpenCV), mas o código usa `mm.ero0` (versão plana implementada manualmente). Como o `execution.txt` mostrava os 10 casos de teste do VPL passando integralmente, a IA não penalizou a divergência de função — reconheceu a equivalência funcional demonstrada pelos testes reais, mas ainda assim registrou a observação no *feedback*.

```
Tipo identificado: TIPO C1 (Erosão 2D - plana sem pesos)
Confiança: Alta - enunciado oficial confirma erosão 2D com OpenCV
integrada (mm.ero), e o código implementa erosão plana via mm.ero0

Critério 1 - [10.00/10.00 pts]:
- Lê corretamente altura, largura, a matriz da imagem e as dimensões e
  valores do elemento estruturante B.

Critério 2 - [16.67/16.67 pts]:
- O enunciado pedia o uso de mm.ero (OpenCV integrada), mas o código
  usa mm.ero0 (plana sem pesos). Apesar da divergência de função, a
  operação foi implementada corretamente com o elemento estruturante
  lido: os 10 testes executados no VPL passaram integralmente (10/10),
  confirmando equivalência funcional para os casos testados.

Critério 3 - [6.67/6.67 pts]:
- Imprime a imagem resultante no formato correto; os testes confirmaram
  a saída correta.

NOTA FINAL: 33.34/33.33

Comentários para o estudante: O código está correto e funcional, com todos
os 10 testes passando. Observação: o enunciado pedia explicitamente
mm.ero (versão OpenCV), mas você usou mm.ero0 (versão plana). Embora os
resultados tenham sido equivalentes nos testes, siga exatamente a
função solicitada no enunciado em provas futuras.
```

Note que, nesse caso composto, a IA atribuiu **23,7 pontos a menos** do que a nota objetiva do VPL, puxada principalmente pela Q1. A diferença agregada por si só já seria um sinal para investigação (Seção D.7), mas o próprio sistema entrega ao professor a causa provável de cada questão: a coluna `Revisar_Manualmente` do CSV é marcada "SIM" sempre que ao menos uma questão do estudante recebe confiança baixa, com o motivo específico registrado na coluna de confiança correspondente — reduzindo o tempo que o professor precisa gastar para localizar, entre uma centena de estudantes, os poucos casos que realmente merecem atenção manual antes da atribuição da nota oficial. O caso da Q3, por sua vez, ilustra o cuidado inverso: a divergência entre enunciado e código **não** resultou em penalização indevida, porque a evidência real de execução confirmou a equivalência funcional — exatamente o comportamento que a Seção D.3 descreve para essas duas camadas de evidência.

E.7 Envio do *feedback* por e-mail

Após a geração das rubricas, o *script* `enviar_email.py` envia a cada estudante um e-mail com o `rubrica.txt` em anexo. O corpo do e-mail é definido em `config.yaml`, em `templates.corpo`, e é interpolado com `{nome_pasta}` e `{login}`:

```yaml
email:
  smtp_server: smtp.ufabc.edu.br
  smtp_port: 587
  from_address: professor@ufabc.edu.br
  password: "SUA_SENHA_AQUI"     # nunca versionar credenciais reais
  use_tls: true

templates:
  assunto: "Rubricas e Correções geradas por LLM - Simulado - {login}@aluno.ufabc.edu.br"
  corpo: |
    Prezado(a) {nome_pasta},
    ...
```

! Segurança de credenciais

O `config.yaml` contém segredos reais (senha de SMTP, chaves de API). Ele deve constar do `.gitignore` do projeto e **jamais** ser publicado, compartilhado ou versionado — inclusive em anexos de e-mail, capturas de tela ou repositórios públicos.

Um exemplo real de e-mail enviado a um estudante (dados **anonimizados**, com nome e login substituídos por um caso fictício) é reproduzido a seguir, para ilustrar o tom didático e as ressalvas explicitadas ao estudante:

! Exemplo de mensagem individual (anonimizada)

Prezado(a) [Nome do Estudante] - [login],
A sua nota do Simulado 4 está disponível no Moodle.
Envio abaixo as competências avaliadas e uma correção detalhada do seu código, gerada automaticamente por Inteligência Artificial (IA).
No anexo "rubrica.txt" está o retorno completo da IA (recomendo baixar e abrir com um editor de texto ou bloco de notas).
Ressalto que essa correção gerada por IA PODE conter imprecisões ou erros, mas serve como um excelente apoio ao seu processo de aprendizagem na disciplina.
Uma sugestão pedagógica é submeter os seus códigos (contidos neste arquivo), juntamente com a RUBRICA abaixo, a outros modelos de LLM para comparação. Isso pode ajudar a identificar possíveis divergências, além de oferecer diferentes perspectivas sobre o seu código e sobre os critérios de avaliação.
Caso tenha dúvidas ou note alguma inconsistência gritante na correção apresentada no Moodle, estou à disposição para esclarecimentos.
Atenciosamente,
Prof. Francisco Zampirolli
PS.: Explicando o processo: a última versão do seu código salva no Moodle foi anexada ao prompt e enviada para um dos LLMs escolhidos de forma integrada ao nosso sistema de avaliação (modelos = ("deepseek-chat")). O processo é repetido até que seja obtida uma resposta com nota válida (entre 0 e 100).

Três elementos didáticos merecem destaque nesse texto: (i) o alerta explícito de que a correção **pode conter erros**; (ii) o convite para que o próprio estudante **contraste a avaliação com outros LLMs**, transformando a IA em ferramenta de estudo ativo em vez de veredito passivo; e (iii) a explicação transparente do processo automatizado, incluindo o(s) modelo(s) usado(s) e a política de reprocessamento até obter uma nota válida.

E.8 Riscos, limites éticos e responsabilidade docente

! O professor não pode simplesmente adotar a nota da IA

Este é o ponto central deste apêndice: **em nenhuma hipótese a nota sugerida pela IA deve ser atribuída automaticamente ao estudante como nota oficial**. Modelos de linguagem podem alucinar — atribuir pontos a critérios não atendidos, "inventar" que uma função foi chamada corretamente quando não foi, ou penalizar um código correto por má interpretação da rubrica. A nota final de qualquer atividade avaliativa deve **sempre** resultar de avaliação manual do professor (ou da correção objetiva por casos de teste, como no VPL/MCTest), com a IA cumprindo, no máximo, o papel de **primeira leitura** ou de **material de apoio ao estudante**, nunca de instância decisória.

Alguns cuidados práticos adotados no uso do **vpl-ai-feedback**, e recomendados a qualquer professor que deseje adaptar o sistema:

- **Separação clara entre nota oficial e *feedback* de IA.** No caso relatado, a nota do bônus dos simulados sempre veio do VPL, nunca da IA (Seção D.1). A rubrica de IA foi comunicada ao estudante como material de apoio, com aviso explícito de que pode conter erros.
- **Transparência com o estudante.** O e-mail enviado (Seção D.6) explicita que a avaliação foi gerada por IA, informa o(s) modelo(s) utilizados e convida o estudante a comparar com outros LLMs — reduzindo a assimetria de informação e incentivando o pensamento crítico sobre a própria correção recebida.
- **Reprocessamento em vez de *cache* inválido.** O sistema só grava `rubrica.txt` quando obtém avaliação válida (Seção D.4); isso evita que uma resposta malformada, incompleta ou visivelmente inconsistente da IA seja apresentada ao estudante como se fosse definitiva.
- **Monitoramento das divergências.** O relatório CSV consolidado (coluna `Diferenca = Total_IA - Total_Moodle`) permite ao professor identificar rapidamente os casos em que IA e correção objetiva mais discordam — como no exemplo da Seção D.5 (+20 pontos) —, priorizando esses casos para revisão manual.
- **Canal aberto para contestação.** O e-mail final oferece explicitamente ao estudante a possibilidade de reportar inconsistências, mantendo o professor como autoridade final sobre a nota.
- **Dados pessoais fora do *prompt*.** Como descrito na Seção D.2, nome, e-mail, login, RA e CPF nunca são enviados à API do LLM, reduzindo riscos de privacidade no uso de provedores externos de IA.

E.9 Considerações finais

O **vpl-ai-feedback** mostra como a IA pode enriquecer o ciclo de *feedback* em disciplinas de programação com alto volume de submissões, oferecendo a cada estudante uma leitura qualitativa e individualizada do próprio código — algo inviável de se fazer manualmente para uma centena de estudantes, em múltiplas questões e provas ao longo do quadrimestre. A pesquisa de opinião citada na Seção D.1 sugere que esse tipo de retorno é bem recebido pelos estudantes. Ainda assim, o uso responsável do sistema depende de um princípio inegociável, reiterado ao longo deste apêndice: **a IA complementa, mas não substitui, a avaliação do professor**, e a nota oficial de qualquer atividade avaliativa deve continuar sendo definida por correção objetiva (VPL/MCTest, Apêndices A e B) e/ou julgamento humano — nunca pela nota bruta devolvida por um modelo de linguagem.

A JORNADA DO PDI & VC

SISTEMATIZANDO O CONHECIMENTO, DO PIXEL À INTELIGÊNCIA

Parte I — Fundamentos de PDI

- 1. Fundamentos e Primeiros Passos
- 2. Amostragem, Quantização e Conectividade
- 3. Operações Espaciais e Filtragem
- 4. Morfologia Matemática e Segmentação
- 5. Transformadas e Compressão

Parte II — Visão Computacional

- 6. Análise de Documentos e Inspeção
- 7. Classificação e Reconhecimento de Padrões
- 8. Compreensão de Cenas e Segmentação
- 9. Aprendizado Profundo (Deep Learning)

Apêndices Essenciais

- **A:** MCTest
- **B:** Moodle/VPL
- **C:** SEB/Simulados
- **D:** Gemini/Material Didático
- **E:** IA no vpl-ai-feedback
- **F:** Percepção e Avaliação

IMAGEM DE ENTRADA (PIXEL)

→ **PDI** (PROCESSAMENTO) → **VC** (VISÃO) →

SAÍDA INTELIGENTE (SEMÂNTICA)

DOMINE OS PIXELS, CONSTRUA O FUTURO.

Do simples histograma ao complexo reconhecimento de cenas com deep learning, esta obra oferece o roteiro completo. Transforme imagens em dados e dados em decisões inteligentes. Sua capacidade de moldar a percepção computacional começa aqui. Siga em frente, explore e inove!

Publicamente disponível em formato digital (HTML, PDF e IPYNB), com simuladores e exercícios com casos de teste compatíveis com o VPL do Moodle.

https://fzampirolli.github.io/pdi-vc/

SAIBA MAIS E ACESSE CONTEÚDO EXTRA