WEBSITE BOOK EDITION

Pixels → Perception

From Pixels to Understanding

Foundations • Vision • Language • Learning • Theory

Volume 1 of 4

Foundations and Image Processing

English Edition • Volume 1

2026-11-05

https://blog.shyandram.dev/

About this edition

This is volume 1 of 4 of the Pixels → Perception course, *Foundations and Image Processing*. It contains 15 published tutorials and 0 blog posts, in the order of the website's learning tracks. This is a typeset edition of the website, not a separately peer-reviewed publication.

The prose, code, equations, and references retain the website's content. Exercise hints are expanded for print. Static figures remain in the book, image comparisons show their input and output images, and interactive demonstrations link to the website. Reference numbers are local to each tutorial.

The website is still under review. AI tools assisted its production, and errors or omissions may remain. Read it alongside the original sources and consult the website for subsequent corrections.

English and Traditional Chinese, and each volume, are separate PDFs. Links to tutorials in this volume navigate within the PDF. Links to tutorials in other volumes, to specific web sections, interactive demonstrations, notebooks, and external sources remain online. Content snapshot: 2026-11-05

Source revision: `2b3c30d`

https://blog.shyandram.dev/

Contents

I Foundations 1

Start Here ·Learning Paths and Recommended Textbooks 2

Images as Arrays 13

II Image Processing 26

IP 01 ·Introduction 27

IP 02 ·Digital Image Fundamentals 45

IP 03 ·Intensity Transformations and Spatial Filtering 67

IP 04 ·Filtering in the Frequency Domain 92

IP 05 ·Image Restoration and Reconstruction 118

IP 06 ·Color Image Processing 144

IP 07 ·Wavelet and Other Image Transforms 167

IP 08 ·Image Compression and Watermarking 188

IP 09 ·Morphological Image Processing 212

IP 10 ·Image Segmentation I: Edge Detection, Thresholding, and Region Detection 236

IP 11 ·Image Segmentation II: Active Contours —Snakes and Level Sets 258

IP 12 ·Feature Extraction 278

IP 13 ·Image Pattern Classification 302

Part I

Foundations

Start Here • Learning Paths and Recommended Textbooks

How the eight tracks fit together, which route to follow for your background and goal, how to run the code and notebooks, and which textbooks to pair with each track.

Online edition: https://blog.shyandram.dev/en/learn/foundations/getting-started/

What you'll learn

- How the site's eight tracks relate to each other and what each one is for.
- How to read a tutorial page: level, prerequisites, the track rail, notebooks and the playground.
- Which learning path fits your background (new to programming, rusty mathematics, already training networks) and your goal (vision, theory, generative models, language, world models).
- How to set up a small Python environment and run the code blocks and notebooks on a CPU.
- Which main textbook to pair with each track, and how to read textbooks, surveys and papers differently.
- How to turn reading into small, checkable milestones.

The big picture

You do not need to finish every mathematical topic before touching an image. You do need to know what each operation assumes. The fastest way to learn is to start from a question you can test: "Why does blur remove an edge?", "Does a classifier still work on images from a new camera?", or "What changes when a model predicts the next token instead of a class?"

> **Explain like I'm five**
>
> Learning this subject is like learning to cook. A textbook explains the ingredients and techniques; a small experiment lets you taste what happens. Pick one dish, change one ingredient at a time, and write down what you notice before you try to cook a whole banquet. This site is the cookbook with the recipes arranged in a sensible order, and every recipe can be cooked in a small kitchen (a laptop CPU).

The site is organized into **eight tracks**. A track is an ordered sequence of tutorials on one theme. The numbering describes the role of each track, not a rule that you must complete one before entering another.

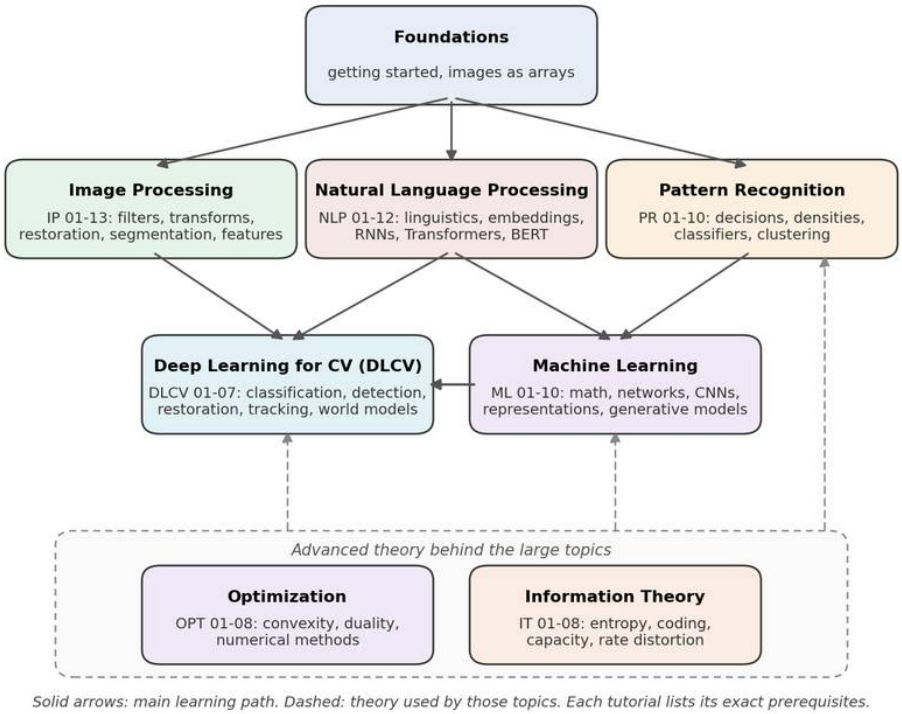


Figure 1 — How the eight tracks fit together. Start with Foundations, then pick an entry track (IP, NLP or PR); Machine Learning and Deep Learning for Computer Vision (DLCV) build on them; Optimization and Information Theory are the advanced theory behind those large topics. Arrows are suggestions, not locks: each tutorial lists its own prerequisites.

Track	Central question	Contents	A useful first outcome
Foundations ·start	What are the data?	this page and Images as Arrays	inspect an image's shape, dtype and coordinate convention
Image Processing (IP) · entry	How can we transform or measure an image?	IP 01–13, from fundamentals to pattern classification	explain a filter and its boundary behavior
NLP ·entry	How are text and language modeled?	NLP 01–12, from the introduction and linguistic essentials through embeddings, RNNs, Transformers and BERT to syntactic and semantic tasks	build and evaluate a small language application
Pattern Recognition (PR) · entry	How can features support decisions under uncertainty?	PR 01–10, from Bayes decisions to a recognition capstone	compare classifiers with a valid train/test split
Machine Learning (ML) · core	How are representations, predictors and generators learned?	ML 01–11, the math chapters ML 01A–01C, and ML 06A on factor models and autoencoders	train and evaluate a small model

Track	Central question	Contents	A useful first outcome
Deep Learning for Computer Vision (DLCV) · core	How do learned models solve visual tasks?	DLCV 01–14: recognition, geometry, restoration, generation, world models and robustness	understand a classification or detection pipeline end to end
Optimization (OPT) · theory	How can we model and solve constrained decisions?	OPT 01–08, from convex sets to solvers and applications	formulate a convex problem and check its optimality conditions
Information Theory (IT) · theory	What are the limits of representing and communicating uncertainty?	IT 01–08, from entropy to coding, channels, rate distortion and networks	calculate entropy and explain a compression or communication limit

How they connect, in one paragraph: **Image Processing** treats an image as a signal and designs operations by hand. **Pattern Recognition** adds statistics and decision theory: given features, how should we decide, and how sure can we be? **ML/DL** replaces hand-designed features with learned ones and supplies the shared machinery (losses, optimization, architectures, generative models) used everywhere else. **DL for Vision** applies that machinery to concrete visual tasks, and its restoration and enhancement chapters lean on the image-formation ideas from Image Processing. **NLP** reuses the same probability, optimization and Transformer ideas with text as input. **Optimization** develops the modeling rules, constraints, duality and numerical methods behind estimation and training. **Information Theory** explains uncertainty, coding and communication limits; its entropy and divergence concepts connect classification, language modeling and compression.

How a tutorial page works

Every tutorial follows the same layout, so once you know one page you know them all.

- **Level pill.** *Beginner*, *intermediate* or *advanced*. The ML math chapters and the first IP tutorials are beginner; the DLCV tutorials assume you already know CNNs and backpropagation.
- **Needs.** Under the title, a *Needs* line links the tutorials that the page assumes. Prerequisites are language independent, so the Chinese and English pages share them. If a *Needs* link is unfamiliar, read that page first, or at least its *Key takeaways*.
- **Track rail.** The left column lists every tutorial in the current track in order; *Previous* and *Next* links at the bottom follow the same order.
- **Sections.** Each tutorial opens with *What you'll learn* and an *Explain like I'm five* box, then develops each topic in two passes: a plain-language statement first, then the precise version with equations and code. It ends with *Key takeaways*, *Exercises* (hints are folded; try before you open them) and numbered *References*, cited in the text as [1], [2] and so on.
- **Notebook buttons.** A tutorial that has a companion notebook shows an **Open in Colab** pill next to its title and a **View notebook** link at the end. Currently ML 03 has one (convolution and image filtering).
- **Interactive labs.** Many pages embed a live experiment that runs a real algorithm on a real image or dataset in your browser: filters, Fourier transforms, thresholds, nearest neighbors and more.

Each lab asks you to predict the outcome first, then lets you check it. All of them also live on the Playground page.

- **Search and language.** The search page finds tutorials and blog posts in the language you are reading; the language switch in the header jumps to the same page in the other language. If a page has not been translated, the Chinese site shows the English version with a notice.

Choose your entry route

Pick the description that fits you best. The routes overlap on purpose.

If programming is new. Learn Python functions, loops, list and array indexing, and basic plotting first. Then read Images as Arrays and IP 01. Make a synthetic image, inspect its rows and columns, and explain why adding two `uint8` images can overflow. Continue with IP 02 and IP 03, which are written so that every equation has a matching NumPy line.

If you can code but mathematics feels rusty. Read ML 01 for the vocabulary of learning, then the three math chapters: linear algebra, probability and information theory and numerical computation. They follow the "applied math" part of Goodfellow, Bengio and Courville [7] and pair each idea with a short NumPy check. Aim to be able to check matrix shapes, interpret a conditional probability and spot a numerically unstable calculation.

If you know introductory probability and linear algebra. Start PR 01 for statistical decisions and classical methods, or ML 02 for learned representations. Return to the math chapters only when a derivation reveals a specific gap.

If you already train neural networks. Skim ML 03 and ML 05, then pick an application chapter in DL for Vision, and pair it with ML 11 on evaluation and deployment.

Suggested learning paths by goal

Once you have an entry point, choose a goal. Each path is a reading order, typically eight to twelve tutorials. Read each page's *Needs* line to fill gaps as they appear.

Goal	Suggested order
Classical image processing	Images as Arrays → IP 01–05 (fundamentals, filtering, frequency domain, restoration) → IP 06–09 (color, wavelets, compression, morphology) → IP 10–13 (segmentation, active contours, features, classification)
Statistical pattern recognition	ML 01A, ML 01B → PR 01 → PR 02 (density estimation) → PR 03 (linear models) → PR 04 (kernels) → PR 05 (features) → PR 07 (clustering, EM) → PR 08 (approximate inference) → PR 09 (ensembles, validation) → PR 10
Deep learning for vision	ML 01 → ML 01A–01C → ML 02 → ML 03 → ML 05 → DLCV 01 → DLCV 02 → DLCV 03 → choose DLCV 09 restoration or DLCV 07 tracking → ML 11

Goal	Suggested order
Generative models	ML 01B → ML 02 → ML 06 → ML 06A → ML 07 → ML 08 → ML 09 → ML 10
Language	NLP 01 → NLP 02 → NLP 03–04 (neural networks, word embeddings) → NLP 05–06 (RNNs, sequence labeling) → NLP 07–09 (attention and Transformers, seq2seq, language modeling) → NLP 10–11 (BERT, classification) → NLP 12; add ML 02 and ML 04 for network and sequence-model foundations
Convex modeling and solvers	ML 01A, ML 01C → OPT 01 → OPT 02–04 (functions, programs, duality) → OPT 05–07 (estimation, numerical methods, constrained solvers) → OPT 08
Information, compression and communication	ML 01B → IT 01 → IT 02–03 (typicality, source coding) → IT 04–06 (capacity, Gaussian channels, rate distortion) → IT 07–08 (statistical decisions, networks)
World models	ML 04 → ML 05 → ML 07 → ML 08 → DLCV 13

A realistic pace is one tutorial per week if you run the code and do two exercises; a long tutorial can take two weeks. Speed matters less than being able to explain each page's key takeaways in your own words.

Run the code

The website and the Python examples use separate environments. You can read the site without installing anything. To run the code you need Python; to preview the website itself you need Node.

A Python environment for the tutorials

Create one dedicated virtual environment for all tutorials. Python 3.11 or newer is a practical starting point; check each package's requirements if you use a different version.

```
python3 -m venv .venv-study
source .venv-study/bin/activate
python -m pip install --upgrade pip
python -m pip install numpy scipy matplotlib scikit-learn scikit-image opencv-python
 ↪  PyWavelets jupyterlab
```

On Windows, activate with `.venv-study\Scripts\Activate.ps1` in PowerShell. These packages cover the introductory array and classical-learning examples. Neural-network chapters also use PyTorch, and specialized examples may name additional dependencies. Install PyTorch with the command produced by its official installation selector [11]; choose the CPU build if you have no GPU. Every example on the site is sized to run on a laptop CPU in seconds to about a minute.

Running a code block

Some examples are self-contained; others deliberately build on earlier blocks in the same chapter. In Images as Arrays, for example, load `gray` and `rgb` in the first block before running the slicing examples. In the PR 10 capstone, run the main workflow in order so later blocks use the original development/test split. Use a fresh notebook kernel for a new chapter, and restart and run all cells before trusting a saved result. An undefined name usually means an earlier setup block has not run; it is not a reason to replace the variable with unrelated data. `assert` statements, where provided, check specific claims; they are not a guarantee that every possible input is supported.

A good habit is to change one thing and re-run: a kernel size, a learning rate, a random seed. The exercises suggest which knob to turn.

Notebooks and Colab

A tutorial with a companion notebook has an **Open in Colab** button. Colab runs the notebook in your browser on a hosted machine, with NumPy, OpenCV, scikit-image and Matplotlib already installed, so nothing needs to be set up locally. To run the same notebook locally, download it from the **View notebook** link, then start `jupyter lab` inside the activated environment and open the file. The notebooks live in the repository's `notebooks/` folder.

Previewing the website (optional)

Only needed if you want to change the site. The repository requires Node 22.12 or newer [12]:

```
npm install
npm run dev
```

The development server prints a local address. Tutorials are MDX files under `src/content/tutorials/<lang>/<track>/`.

Your first CPU experiment

This complete example creates its own image, adds noise and smooths it. No download is needed.

```
import numpy as np
import cv2

rng = np.random.default_rng(0)
clean = np.zeros((32, 32), dtype=np.float32)
clean[8:24, 8:24] = 1.0
noisy = clean + rng.normal(0, 0.1, clean.shape).astype(np.float32)
smooth = cv2.GaussianBlur(noisy, (5, 5), 1.0,
```

```
                          borderType=cv2.BORDER_REFLECT_101)
print("Shape and type:", smooth.shape, smooth.dtype)
print("Noise-region standard deviation:",
      noisy[:5, :5].std(), smooth[:5, :5].std())
print("Across the edge:", np.round(smooth[16, 5:12], 3))
assert smooth.shape == clean.shape
assert np.isfinite(smooth).all()
```

The output shows two effects. The standard deviation in the flat, all-background corner drops from about 0.074 to about 0.026: smoothing averages away noise. Across the edge at column 8, the values now ramp from about 0.04 to 0.93 over three or four pixels instead of jumping from 0 to 1: smoothing also spreads a sharp transition. A lower noise estimate therefore does not mean every important detail improved. IP 03 explains why, and Exercise 2 asks you to measure the trade-off.

Choose one main textbook

Use one main book for continuity and a second one for a different explanation of the same idea. The links point to author or publisher pages, with access conditions as described there. This is a curated starting set, not a ranking.

Need	Recommended book	How to use it
Mathematical preparation	Deisenroth, Faisal and Ong, *Mathematics for Machine Learning* [1]	start with linear algebra, matrix decompositions, vector calculus and probability; apply them through regression and PCA
Image operations	Gonzalez and Woods, *Digital Image Processing*, 4th ed. [2]	pair image fundamentals, spatial and frequency filtering, restoration and segmentation with the DIP track
Pattern Recognition, main text	Theodoridis and Koutroumbas, *Pattern Recognition*, 4th ed. [3]	follow classifier design, features and clustering; work the problems rather than only reading summaries
Pattern Recognition, companion	Bishop, *Pattern Recognition and Machine Learning* [4]	revisit the same topics through probability, latent variables and graphical models
A first modern DL text	Prince, *Understanding Deep Learning* [5]	connect diagrams, mathematics and the accompanying notebooks for networks and generative modeling
Another modern DL explanation	C. M. Bishop and H. Bishop, *Deep Learning: Foundations and Concepts* [6]	use its probabilistic presentation to strengthen the concepts behind architectures and training
DL foundations and reference	Goodfellow, Bengio and Courville, *Deep Learning* [7]	Part I (applied math) matches ML 01A–01C; later chapters cover optimization, CNNs, sequence models and representation learning
Computer vision beyond classification	Szeliski, *Computer Vision: Algorithms and Applications*, 2nd ed. [8]	connect image formation, motion, geometry, recognition and reconstruction

Need	Recommended book	How to use it
Probabilistic machine learning	Murphy, *Probabilistic Machine Learning: An Introduction* [9]	a modern, probability-first reference for the ML and PR tracks
NLP applications, main text	Hagiwara, *Real-World Natural Language Processing* [13]	follow the application workflow, text classification, sequence models, attention, pretrained models, training practices and deployment
NLP linguistic foundations, companion	McRoy, *Principles of Natural Language Processing* [14]	study syntax, grammars, semantics, discourse, retrieval, generation and translation alongside the corresponding NLP chapters
Language and text, supplementary	Jurafsky and Martin, *Speech and Language Processing*, 3rd-edition draft [10]	deepen selected topics in tokens, n-grams, classification, embeddings, Transformers and evaluation
Convex optimization	Boyd and Vandenberghe, *Convex Optimization* [15]	OPT 01–05 follow modeling and applications in Chapters 1–8; OPT 06–08 develop numerical methods in Chapters 9–11 and modern extensions
Information theory	Cover and Thomas, *Elements of Information Theory*, 2nd ed. [16]	follow entropy and typicality through source/channel coding, continuous variables, rate distortion, statistical decisions, universal coding and networks

For the Pattern Recognition track, **Theodoridis and Koutroumbas [3] and Bishop's PRML [4] are the two core references**. Read Theodoridis for a systematic recognition pipeline, then compare Bishop's probabilistic treatment of the same problem. Their publication dates do not weaken Bayes decision theory or Gaussian models; turn to newer papers when the architecture, data scale or application has changed.

If you choose just one deep learning introduction, start with Prince [5] and use this site's math chapters as needed. Choose Bishop and Bishop [6] if you prefer a sustained probabilistic viewpoint. Goodfellow et al. [7] remains a valuable foundation, but its 2016 edition predates Transformers in vision, diffusion models and foundation models; the ML track covers those.

The NLP track uses **Hagiwara [13] and McRoy [14] as its two core books**. Hagiwara connects models to applications; McRoy supplies the linguistic and symbolic concepts needed to understand what a language system is trying to represent. Read both around the same problem: a classifier needs a data workflow, but a parser or dialogue system also needs an explicit account of linguistic structure. Newer papers supplement their 2021 editions.

For Optimization, use **Boyd and Vandenberghe [15]** to learn which problem transformations preserve convexity and what an optimality certificate means. For Information Theory, use **Cover and Thomas [16]** to distinguish an achievable construction from a bound that no construction can beat. These tracks extend the introductory mathematics in ML 01B–01C; their book chapter maps appear within each tutorial.

The Jurafsky and Martin draft changes over time; its current release is dated August 19, 2026 [10]. Record the release date when you make chapter notes, because chapter numbering can

move between versions. Likewise, record textbook edition numbers so that exercise references stay reproducible.

Read textbooks, surveys and papers differently

A **textbook** gives definitions and a coherent sequence. Re-derive one equation, run a tiny example and solve one exercise before moving on.

A **survey or review** gives a map of alternatives, assumptions and open questions. Each tutorial's *Modern view* section summarizes several. When you read one, make a table with the prediction target, learning objective, required data, compute and evaluation of each method it covers. Check its coverage date: "recent" means recent to its authors.

An **original paper** supplies one method and its evidence. Read the problem, the method, the experimental protocol and the limitations. Trace an important claim to the actual table or ablation that supports it. Knowing what changed relative to a fair baseline is more useful than knowing the method's name.

Keep a reading note with five entries: the question, the mathematical objects, the main assumption, the experiment that tests it, and one remaining doubt. The reference lists in each tutorial suggest focused next readings; you do not need to read every cited paper before running the code.

A sequence of small milestones

1. **Represent data:** inspect an image and explain its axes, range and dtype.
2. **Transform data:** implement one filter, compare it with a library function and account for the boundaries.
3. **Make a decision:** fit a simple classifier with preprocessing learned only from the training data.
4. **Learn features:** train a small neural network and justify its loss and validation choice.
5. **Study a modern method:** reproduce a mechanism such as attention, masking or noising on synthetic data.
6. **Finish a small project:** report a baseline, independent test results, failure cases and limitations.

Move on when you can explain the previous milestone without copying its wording. A modest experiment with a clean comparison teaches more than a large model whose data and evaluation you cannot inspect.

Key takeaways

- Eight tracks: Foundations, Image Processing, NLP, Pattern Recognition, ML/DL, DL for Vision, Optimization and Information Theory. They feed each other; none has to be finished before another is started.

- Each page's *Needs* line, level pill and track rail tell you where it sits; use them instead of reading everything in order.
- Choose an entry route by background and a learning path by goal; one tutorial per week with code and exercises is a sustainable pace.
- Examples are designed for CPU study. Run dependent blocks in chapter order, keep each chapter in its own notebook, and install additional packages when a specific example requires them.
- Pair each track with one main textbook and one companion, and read textbooks, surveys and papers for different purposes.

Exercises

1. **Plan.** Pick an entry route, a learning path and a main textbook. Write the question your first experiment will answer and a concrete result that would count as progress.

   Hint

   Keep the question small enough to test on a CPU in one sitting, for example "How much does a 5×5 Gaussian blur reduce noise in flat regions, and how much does it widen an edge?" Progress is a number you can measure, not a feeling.

2. **Kernel sweep.** Change the smoothing example to kernel sizes 3, 5 and 9 (raise σ with the kernel, for example 0.8, 1.0 and 2.0). Report the flat-region standard deviation and the number of pixels the edge takes to rise from 0.1 to 0.9.

   Hint

   Keep the random seed and the noisy image fixed so only the filter changes. Noise should fall and the edge width should grow together: that is the bias–variance trade-off of linear smoothing.

3. **Read a page's metadata.** Open ML 02. What is its level, what does its *Needs* line point to, and which tutorial comes next in the track rail?

   Hint

   The *Needs* line is right under the title. Follow it back until you reach a page whose prerequisites you already know; that chain is your personal reading order.

4. **Classify a source.** Select one reference from any tutorial and classify it as textbook, survey, original paper or software documentation. What question is that source best suited to answer?

   Hint

   A survey answers "what are the options and how do they compare?"; an original paper answers "does this specific change help, and under what protocol?"; documentation answers "what does this function actually compute?"

5. **Split by the right unit.** Sketch a training/validation/test split for a dataset containing many frames from each of 50 videos. Explain why splitting individual frames at random can be misleading.

Hint

Neighboring frames are nearly identical, so a random frame split puts near-duplicates in both training and test sets and inflates the test score. Split by video (or by recording session) instead.

References

1. M. P. Deisenroth, A. A. Faisal and C. S. Ong, *Mathematics for Machine Learning*, Cambridge University Press, 2020. Book site
2. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018. Book site
3. S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, 4th ed., Academic Press, 2009. Companion site
4. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006 (official PDF available). Book page
5. S. J. D. Prince, *Understanding Deep Learning*, MIT Press, 2023. Book and notebooks
6. C. M. Bishop and H. Bishop, *Deep Learning: Foundations and Concepts*, Springer, 2024 (free online version). Book site
7. I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, MIT Press, 2016. Book site
8. R. Szeliski, *Computer Vision: Algorithms and Applications*, 2nd ed., Springer, 2022 (PDF for personal use). Book site
9. K. P. Murphy, *Probabilistic Machine Learning: An Introduction*, MIT Press, 2022. Book site
10. D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 3rd ed. draft, release of August 19, 2026. Draft site
11. PyTorch, "Get Started Locally," installation documentation. Documentation
12. Node.js, "Download Node.js." Download page
13. M. Hagiwara, *Real-World Natural Language Processing*, Manning, 2021. Publisher
14. S. McRoy, *Principles of Natural Language Processing*, open textbook, 2021. Open textbook
15. S. Boyd and L. Vandenberghe, *Convex Optimization*, Cambridge University Press, 2004. Author site and book
16. T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed., Wiley, 2006. Publisher

Links and draft dates were checked on October 4, 2026. A companion website does not imply that the complete textbook is free; access conditions are those stated on each site.

Images as Arrays

What a digital image really is, how NumPy and OpenCV store it, and the indexing, dtype and channel-order habits that prevent the most common bugs.

Online edition: https://blog.shyandram.dev/en/learn/foundations/images-as-arrays/

What you'll learn

- How a grayscale or color image becomes a NumPy array, and what `shape`, `dtype` and `strides` tell you about it.
- Why an image is a *sampled and quantized* version of a continuous scene, and what is lost when you keep fewer pixels or fewer gray levels.
- The coordinate convention (row first, then column), and how slicing, flipping, masking and flattening work, including the difference between a view and a copy.
- The three pitfalls that bite everyone: BGR versus RGB channel order, `uint8` wraparound, and writing through a view by accident.
- How the same array view of an image carries over to deep-learning tensors, image patches and neural image representations.

The big picture

Every method in this site, from a 3×3 blur to a vision Transformer, starts from the same object: a grid of numbers. If you can read, slice and transform that grid with confidence, the rest of the curriculum is a matter of choosing which numbers to combine and how.

> **Explain like I'm five**
>
> Picture a mosaic made of tiny colored tiles. Stand far away and you see a face; stand close and you see only squares, each with one color. A digital photo is the same: a rectangle of tiles, and every tile is described by a number (or three numbers, one each for red, green and blue). To find a tile you say "which row, which column", the way you find a seat in a cinema. Computers store the mosaic as a table of numbers, and everything you do to a picture, such as brightening it, blurring it or finding faces in it, is arithmetic on that table.

Why it matters: almost every beginner bug in image code is really a bug about the array, not about the algorithm. Colors look blue because the channels were swapped. Bright areas turn black because 8-bit values wrapped around. An edit changes the original because a crop was a view, not a copy. This page names those traps once, with code you can run, so you can recognize them later.

An image is a sampled, quantized signal

Plain version. The real world has no pixels. A camera measures how much light reaches each tiny spot of its sensor, and writes the measurement down as a whole number. Two decisions turn a continuous scene into a digital image: *where* to measure (how many spots) and *how finely* to write each measurement (how many possible values).

Precise version. Let a monochrome scene be a continuous brightness function. **Sampling** evaluates it on a regular grid of H rows and W columns. **Quantization** rounds each sample to one of $L = 2^b$ integer levels, where b is the number of bits per pixel. The result is a function on a finite grid,

$$I : \{0, \dots, H-1\} \times \{0, \dots, W-1\} \to \{0, \dots, L-1\}, \qquad (r, c) \mapsto I[r, c],$$

where r is the **row** index (counted from the top), c is the **column** index (counted from the left), $I[r, c]$ is the **pixel value**, and $L = 256$ for the common 8-bit case, so values run from 0 (black) to 255 (white). A color image adds a channel index k and becomes $I[r, c, k]$ with $k \in \{0, 1, 2\}$ for the three color channels. This is the standard treatment of image formation and digitization in textbooks such as Gonzalez and Woods [1, Ch. 2] and Torralba, Isola and Freeman [2].

Written as an array, the grid has a **shape** `(H, W)` for grayscale or `(H, W, 3)` for color. The number of values stored is $H \times W \times C$ for C channels, and the memory needed is that count times the bytes per value.

> **Coordinates: row first**
>
> Arrays are indexed `[row, column]`, which is `[y, x]` in the picture's own axes: the first index runs down the page, the second runs across. Graphics functions often take points as `(x, y)`, which is `(column, row)`. This series' image-processing chapters follow the textbook convention and write $f(x, y)$ with x as the row index, so $f(x, y)$ matches NumPy's `f[x, y]`. Whenever code and math disagree, check which axis is which before debugging anything else.

Shape, dtype and memory

Plain version. An array knows three things about itself: how many numbers it holds along each axis (`shape`), what kind of number each is (`dtype`), and how those numbers are laid out in memory (`strides`).

scikit-image ships a few small test photographs [6], [7], which keeps every example here runnable without downloading anything. The code below loads one grayscale and one color image and prints their basic properties.

```
import numpy as np
from skimage import data

gray = data.camera()          # grayscale test photo bundled with scikit-image
rgb = data.astronaut()        # color test photo, also bundled
```

```
for name, a in [("gray", gray), ("rgb", rgb)]:
    print(name, a.shape, a.dtype, a.ndim, a.size, a.nbytes)
print(gray.min(), gray.max())
print(gray.strides, rgb.strides)
```

Output:

```
gray (512, 512) uint8 2 262144 262144
rgb (512, 512, 3) uint8 3 786432 786432
0 255
(512, 1) (1536, 3, 1)
```

Reading the output: the grayscale image has 512 rows and 512 columns, so `size` is $512 \times 512 = 262{,}144$ values, and since each `uint8` takes one byte, `nbytes` is the same number. The color image has three times as many values. The pixel range is the full 0 to 255.

`strides` is the number of bytes to jump to move one step along each axis. For `gray`, moving one column costs 1 byte and moving one row costs 512 bytes, which means NumPy stores the image **row by row** (C order). In `rgb`, the three channel values of one pixel sit next to each other in memory. This explains why scanning along a row is fast and why reshaping an image into a long vector keeps rows together, as the next section shows. The NumPy paper [3] and documentation [4] describe the array model in full.

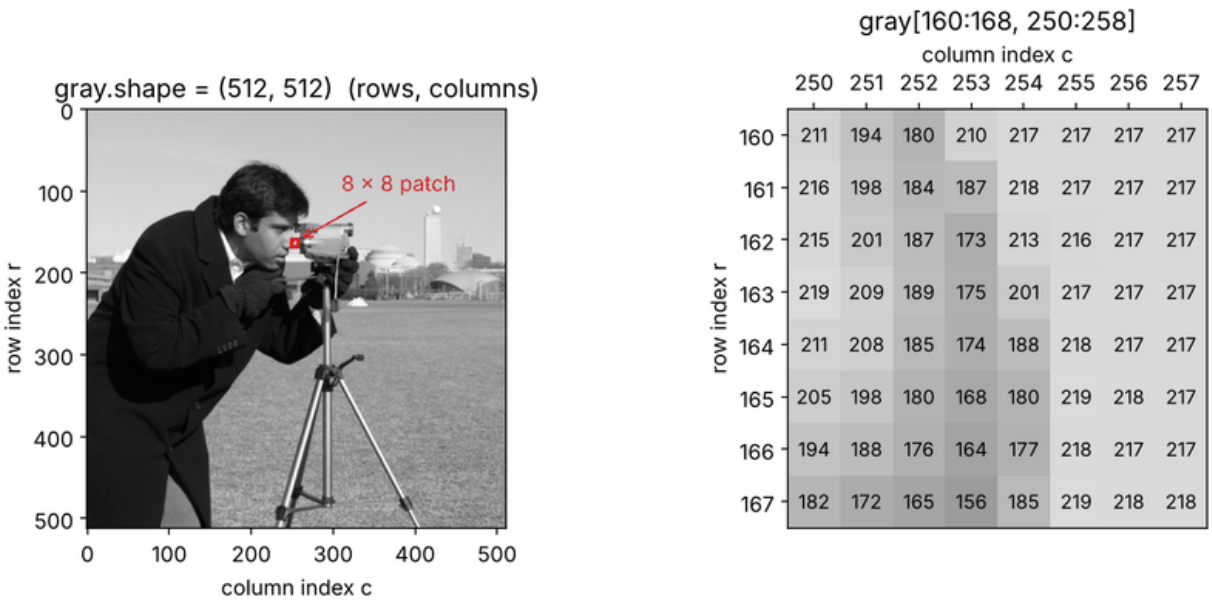


Figure 1 — An image is a table of integers. Left: the 512 × 512 test photo with an 8 × 8 patch outlined. Right: the same patch as the numbers the computer stores, with its row and column indices. Matplotlib places row 0 at the top by default [9].

Indexing, slicing and views

Plain version. Because an image is just an array, you pick out pixels, rectangles and whole rows with the same square-bracket syntax you use for any NumPy array. The surprise is that a rectangle taken this way is a *window onto the same memory*, not a separate copy.

Precise version. For an array `a`, the expression `a[r, c]` returns one pixel, `a[r0:r1, c0:c1]` returns the sub-block of rows $r_0, \dots, r_1 - 1$ and columns $c_0, \dots, c_1 - 1$ (the end index is excluded), and a step such as `a[::2, ::2]` keeps every second row and column. Basic slices are **views**: array objects sharing the original memory. Integer indexing that selects every axis, such as `gray[r, c]`, returns a scalar value instead; `rgb[r, c]` leaves the channel axis and returns a view. Boolean or integer-array indexing returns a copy when reading, but direct indexed assignment such as `a[mask] = 0` still writes into `a`. Calling `.copy()` explicitly separates a slice from its source [13].

```
r, c = 100, 200
print(gray[r, c])             # row first, then column
print(rgb[r, c])              # three numbers: R, G, B
patch = gray[100:108, 200:208]            # rows 100..107, columns 200..207
print(patch.shape, np.shares_memory(patch, gray))

work = gray.copy()
work[100:108, 200:208] = 0                # writing through a slice edits `work`
print(work[100:108, 200:208].max(), gray[100:108, 200:208].max())

H, W = gray.shape
vec = gray.reshape(-1)                    # flatten: row-major order
print(vec.shape, vec[r * W + c] == gray[r, c], np.shares_memory(vec, gray))
print(np.array_equal(gray[::-1], np.flipud(gray)), gray[:, ::-1].shape)
mask = gray > 200
print(mask.dtype, mask.sum())
```

Output:

```
54
[81 57 17]
(8, 8) True
0 109
(262144,) True True
True (512, 512)
bool 55112
```

What each line shows:

- `gray[100, 200]` is 54 and `rgb[100, 200]` is `[81 57 17]`: a color pixel is a short vector, one number per channel.
- The 8×8 patch shares memory with `gray`, so it is a view. Because `work` is a copy of `gray`, zeroing a patch of `work` leaves `gray` untouched (its patch still has maximum 109). If you had written to the view of `gray` itself, the original photo would have changed.

- Flattening gives a vector of $HW = 262144$ numbers in which pixel (r, c) sits at position $rW + c$. This flattened form is the input vector that classical classifiers and the first layer of a fully connected network see (see ML 01A and ML 01).
- Flipping the image upside down is `gray[::-1]`, a view with a negative row step; no loop is needed. The comparison with `np.flipud` is `True`.
- A comparison such as `gray > 200` yields a boolean array of the same shape, here with 55,112 `True` entries. Boolean masks are how you select, count or edit pixels that satisfy a condition.

Color images and channel order

Plain version. A color image stores three numbers per pixel, one for how red, one for how green and one for how blue. The trouble is that two common conventions order these three numbers differently, and nothing in the array tells you which one you have.

Precise version. Most Python imaging code (NumPy arrays from scikit-image [6], Matplotlib, Pillow) uses the order red, green, blue (**RGB**). OpenCV's `imread` returns **BGR**: blue is channel 0 and red is channel 2. Mixing the two does not raise an error; it silently swaps red and blue, so skin looks blue and the sky looks orange. To make the effect visible without a file on disk, the code below writes the color test image to a temporary PNG, reads it back with OpenCV and compares.

```
import pathlib, tempfile
import cv2
from skimage import io

with tempfile.TemporaryDirectory() as d:
    path = pathlib.Path(d) / "astronaut.png"
    io.imsave(path, rgb)                    # a PNG written in R, G, B order
    bgr = cv2.imread(str(path))             # OpenCV hands the channels back as B, G, R
print(rgb[100, 200], bgr[100, 200])
print(np.array_equal(bgr[..., ::-1], rgb))
print(np.array_equal(cv2.cvtColor(bgr, cv2.COLOR_BGR2RGB), rgb))

weights = np.array([0.299, 0.587, 0.114])
manual = rgb @ weights                      # weighted sum of R, G, B -> float64 (H, W)
cv_gray = cv2.cvtColor(rgb, cv2.COLOR_RGB2GRAY)
print(cv_gray.dtype, np.abs(manual - cv_gray).max())
from skimage.color import rgb2gray
print(np.abs(rgb2gray(rgb) * 255 - cv_gray).max())
```

Output:

```
[81 57 17] [17 57 81]
True
True
uint8 0.5010000000000332
15.529299999999992
```

The same pixel reads `[81 57 17]` as RGB and `[17 57 81]` as BGR. Reversing the last axis, `bgr[..., ::-1]`, is a free way to flip the order, and `cv2.cvtColor` with `COLOR_BGR2RGB` does the same thing explicitly. Both restore the original exactly.

To turn color into gray, the standard approach is a **weighted sum** of the channels, because the eye is more sensitive to green than to blue. The weights $0.299, 0.587, 0.114$ come from the luma definition in the ITU-R BT.601 television standard [8, Sec. 2.5.1], and the check above shows OpenCV's gray conversion agrees with that weighted sum to within 0.501 gray level, which is just rounding to integers. Different libraries use different weights: scikit-image's `rgb2gray` differs from OpenCV's result by up to about 15.5 gray levels on this image. So never assume that "gray" means the same numbers across libraries. Compare against the documentation of the function you call.



Figure 2 — A color image is a stack of three planes. Each channel is itself a grayscale image: the orange suit is bright in the red plane and dark in the blue plane. The last panel is the weighted gray conversion.

Data types and the wraparound trap

Plain version. The most common storage type for images is the 8-bit unsigned integer, which can only hold 0 to 255. If a calculation goes past 255 it does not stop at 255: it silently starts again from 0. A pixel that was almost white can become almost black.

Precise version. For an unsigned integer type `uint8`, arithmetic is performed modulo $2^8 = 256$. So $250 + 10 = 260 \equiv 4$ and $250 - 251 = -1 \equiv 255$. Because values are not saturated, brightening or subtracting images in `uint8` is unsafe. NumPy 2 also refuses to combine an array with a Python integer that does not fit the array's type: it raises an error instead of quietly widening, following NEP 50 [5]. The safe recipe has three steps: convert to a wider type (`int16`, `float32`), compute, then clip back to the valid range and convert back to `uint8`.

```
from skimage.util import img_as_float

a = np.array([250], dtype=np.uint8)
print(a + 10, a - 251, a.astype(np.int16) + 10)
try:
    a + 300
except OverflowError as e:
    print("OverflowError:", e)
```

```
wrapped = gray + np.uint8(60)
clipped = np.clip(gray.astype(np.int16) + 60, 0, 255).astype(np.uint8)
print((gray >= 196).sum(), (wrapped < gray).sum(), clipped.max())

x = gray.astype(np.float32) / 255.0
print(x.dtype, x.min(), x.max(), np.allclose(x, img_as_float(gray)))
y = np.clip(1.2 * x, 0, 1)
out = np.round(y * 255).astype(np.uint8)
print(out.dtype, out.max())
```

Output:

```
[4] [255] [260]
OverflowError: Python integer 300 out of bounds for uint8
72087 72087 255
float32 0.0 1.0 True
uint8 255
```

Reading it: in `uint8`, `250 + 10` gives 4 and `250 - 251` gives 255, while in `int16` the sum is a correct 260. Adding 60 to every pixel wraps exactly the pixels whose value is at least 196 (since $196 + 60 = 256$): there are 72,087 of them, and the same 72,087 pixels come out *darker* than they went in. The clipped version keeps all of them at the maximum 255 instead. The last block shows the usual float workflow: divide by 255 to work in $[0, 1]$, compute, clip, multiply back and round. `skimage.util.img_as_float` performs the same rescaling for `uint8` images.

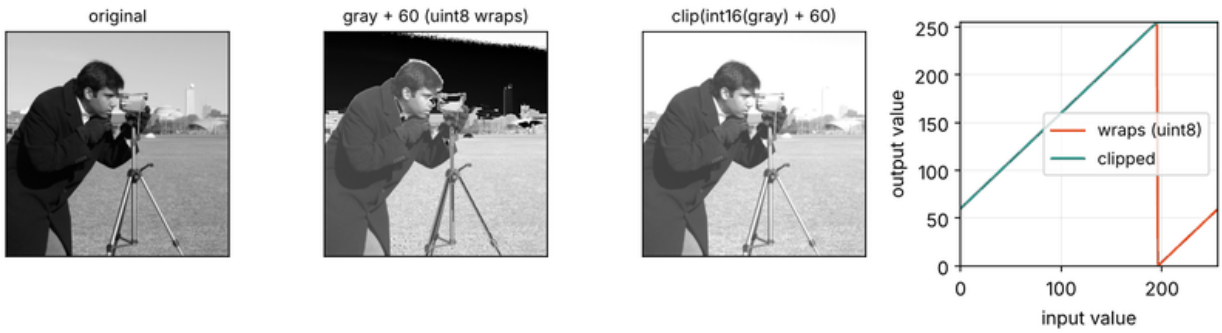


Figure 3 — Brightening by 60 levels. In uint8 the bright sky wraps around to black (orange curve). Converting to a wider type, adding, and clipping saturates it to white instead (green curve).

Interactive demonstration (online): open the online edition of this tutorial to use this demonstration.Open demonstration

Choose the dtype before the arithmetic

Casting after an overflow cannot recover the lost value. Decide the working type first: `float32` in $[0, 1]$ for filtering and learning, `int16` for exact integer differences, and convert back to `uint8` only for

saving or display. OpenCV's own arithmetic helpers such as `cv2.add` saturate instead of wrapping, but NumPy's + operator does not, so know which one a line of code is using. The lab above runs the same arithmetic on the photo, so you can try each choice yourself.

Sampling and quantization on a real image

Plain version. Two independent knobs control how much of the scene survives: the number of pixels (spatial resolution) and the number of gray levels (brightness resolution). Turning down the first makes the picture blocky; turning down the second makes smooth shading break into flat bands.

Precise version. Keeping every s-th row and column, `gray[::s, ::s]`, reduces an $H \times W$ image to about $H/s \times W/s$ samples. Done without smoothing first, this can create **aliasing**: fine detail that the coarser grid cannot represent shows up as false patterns instead of disappearing (the formal treatment is in IP 02 and the frequency-domain chapters). Reducing the bit depth from 8 to b keeps only 2^b distinct values. The helper below rounds each pixel to the center of its bin, so with 4 bits there are 16 allowed gray levels.

```
def quantize(img, bits):
    step = 256 // 2 ** bits
    return (img // step) * step + step // 2    # replace each bin by its center

for bits in (8, 4, 2, 1):
    print(bits, len(np.unique(quantize(gray, bits))))

small = gray[::4, ::4]                          # keep every 4th row and column
print(gray.shape, "->", small.shape, small.nbytes, gray.nbytes)

def invert_loop(img):
    out = np.empty_like(img)
    for r in range(img.shape[0]):
        for c in range(img.shape[1]):
            out[r, c] = 255 - img[r, c]
    return out

print(np.array_equal(invert_loop(gray), 255 - gray))
chw = np.transpose(rgb, (2, 0, 1))
print(rgb.shape, "->", chw.shape, np.shares_memory(chw, rgb))
print(np.mean(rgb, axis=(0, 1)))
```

Output:

```
8 256
4 16
2 4
1 2
(512, 512) -> (128, 128) 16384 262144
```

```
True
(512, 512, 3) -> (3, 512, 512) True
[141.56249237 105.75944519  96.47507477]
```

The number of distinct values falls as 2^b. Keeping every 4th row and column shrinks the image from 262,144 bytes to 16,384, one sixteenth of the memory. The nested loop and the single expression `255 - gray` give identical results (`True`), but the array expression runs as one compiled operation rather than a quarter of a million Python steps: write image code as array expressions whenever you can, and keep explicit loops for checking. Transposing to channels-first, `(3, 512 , 512)`, is another view, which is the layout most deep-learning libraries expect. The last line, a mean over both spatial axes, gives one average per color channel.

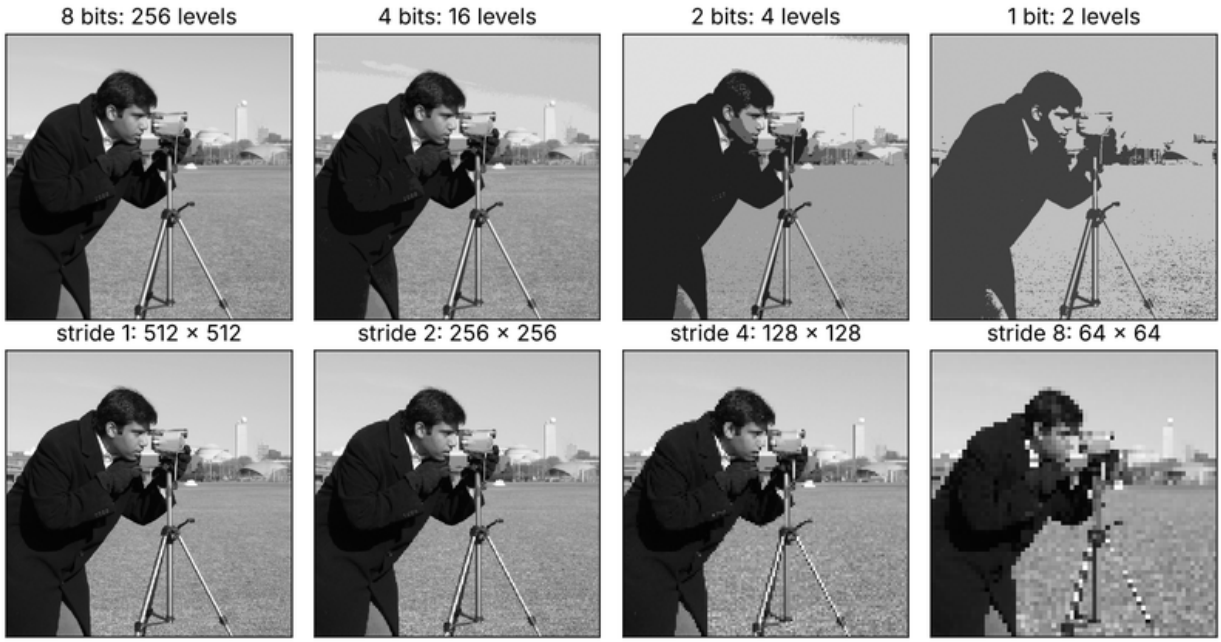


Figure 4 — Two ways to lose information. Top: fewer gray levels (quantization) turns smooth shading into flat bands. Bottom: fewer samples (here plain point sampling, shown at equal size) makes the image blocky and breaks thin structures such as the tripod legs.

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

Eight bits per channel is usually enough to show an image, but not always enough to process one: stretching the contrast of a dark, banded image only magnifies the bands. That is why image formats and sensors with more bits per channel exist, and why processing pipelines convert to floating point for the arithmetic. The lab above moves both knobs on a photo and on a test pattern that exposes aliasing.

A short checklist before you process an image

Check	Why	One line
Print `shape` and `dtype`	Confirms grayscale versus color and the value range	`print(a.shape, a.dtype, a.min(), a.max())`
Know the channel order	RGB and BGR look similar until the colors are wrong	`rgb = bgr[..., ::-1]`
Choose the working dtype	`uint8` wraps; `float32` in $[0, 1]$ is safe for filtering	`x = a.astype(np.float32) / 255`
Copy before editing	Slices are views and edits write through	`b = a.copy()`
Clip before converting back	Out-of-range floats become garbage in `uint8`	`np.clip(y, 0, 1)`
Remember `[row, column]`	Pairs from graphics APIs are `(x, y)`	`a[y, x]`

Modern view

Nothing in this chapter is obsolete, because deep-learning frameworks treat an image as a tensor and tensors behave like NumPy arrays. What changes is the *layout*, the *normalization* and, in recent research, the idea of what an image is.

Tensors and layout. A training batch adds a leading axis, giving `(N, H, W, C)` or the channels-first `(N, C, H, W)` that many libraries expect, as in the `np.transpose` view above. The pixel values are usually converted to `float32` and normalized, for example by dividing by 255 and standardizing each channel. All the pitfalls above carry over unchanged: a model trained on RGB input and fed BGR will silently degrade, and a preprocessing step that wraps in `uint8` can poison a whole dataset without raising an error. The convolution and Transformer chapters build directly on this view: ML 03 and ML 05.

Images as sequences of patches. The Vision Transformer of Dosovitskiy et al. [10] (ICLR 2021) cuts an image into a grid of fixed-size patches, flattens each patch into a vector with the reshaping used in this chapter, and feeds the sequence to a standard Transformer. The paper reports that such a model, with large-scale pre-training, can match or exceed convolutional networks on image classification benchmarks while needing less training compute. The key point for this chapter is that the array operations (slicing into blocks, reshaping, transposing) *are* the model's front end.

Images as functions. A second modern idea replaces the grid with a learned function. Sitzmann et al. [11] (NeurIPS 2020) propose sinusoidal representation networks, small networks with periodic activations that take coordinates (r, c) as input and return the pixel value, and show that they can represent images, audio, video and their derivatives in detail. Essakine et al. [12] (TMLR 2025) survey this *implicit neural representation* field: they propose a taxonomy over four groups of methods (activation functions, position encoding, combined strategies and network-structure optimization), compare them experimentally, and conclude with open problems such as more expressive activation functions, better position encodings and scalability to high-dimensional data. The array is still how such models are trained and evaluated, since the network is fitted to sam-

pled pixels; but resolution becomes a property of how you query the function, not of the stored table.

What is still open and what is not. Questions that remain unsettled include how to store high-dynamic-range and wide-gamut color consistently across tools, and how learned representations should be compared with plain arrays at the same storage cost. Questions that are settled, and worth drilling until they are automatic, are the ones in this chapter: shape, dtype, channel order, views versus copies, and the habit of choosing the arithmetic type before computing.

Key takeaways

- A digital image is a sampled, quantized signal stored as an array of shape `(H, W)` or `(H, W, C)`; $I[r, c]$ means row r, column c.
- `shape`, `dtype` and `strides` describe the array. A 512×512 8-bit grayscale image holds 262,144 bytes; the color version holds three times as many.
- Slicing returns views that share memory with the original. Use `.copy()` before editing, and use array expressions rather than pixel loops.
- Libraries disagree on channel order (OpenCV uses BGR, most others RGB) and on the weights used for gray conversion; check, do not assume.
- `uint8` arithmetic wraps modulo 256. Convert to `int16` or `float32`, compute, clip, and only then convert back.
- Spatial resolution and gray-level resolution are separate choices, and reducing either loses information in a different way: blockiness and aliasing versus banding.
- Deep-learning tensors, image patches and implicit neural representations are all built on the same array view of an image.

Exercises

1. **Channel means.** Load `data.astronaut()` and compute the mean of each color channel with `rgb.mean(axis=(0, 1))`. Which channel is largest? Then write the result for the BGR version of the same image and explain which index now holds the largest value.

   Hint

   For this image the means are about 141.6, 105.8 and 96.5 for R, G and B, so red is largest. In the BGR array the same numbers appear in reversed order, so the largest value is at index 2 instead of index 0.

2. **Flip with slicing.** Using only slicing (no `np.flip` and no loops), produce the upside-down, left-right mirrored and 180°-rotated versions of `gray`. Check each against `np.flipud`, `np.fliplr` and `np.rot90(gray, 2)`.

   Hint

   Use a step of -1 on the axis you want to reverse: `gray[::-1]`, `gray[:, ::-1]`, and `gray[::-1, ::-1]`. All three are views, so copy them if you plan to edit.

3. **Find the wrap.** For the brightening by 60 in this page, which input gray values wrap around, and what output value does an input of 200 produce? Write a one-line check that counts the wrapped pixels without computing the wrapped image.

   Hint

   Inputs of 196 or more wrap, because $196 + 60 = 256 \equiv 0$. An input of 200 gives $260 \bmod 256 = 4$. The count is `(gray >= 196).sum()`.

4. **Safe difference.** Two `uint8` frames A and B should be compared with the absolute difference $|A - B|$. Show with a pair of pixel values that `A - B` fails, and write a safe version in NumPy.

   Hint

   With $A = 10$ and $B = 20$ in `uint8`, `A - B` equals 246, not 10. A safe version is `np.abs(A.astype(np.int16) - B.astype(np.int16)).astype(np.uint8)`.

5. **Design a bit-depth budget.** A grayscale image is stored with 8 bits, then quantized to b bits. Predict the number of distinct values and the memory needed if each pixel is packed to exactly b bits, for $b = 4$ and $b = 2$ at 512×512. Why does the picture in Figure 4 show bands at 4 bits but not at 8?

   Hint

   There are 2^b values: 16 and 4. Packed storage is $512 \times 512 \times b/8$ bytes, which is 131,072 bytes at 4 bits and 65,536 bytes at 2 bits. Bands appear where a smooth gradient crosses a boundary between two bins: at 8 bits the steps are 1 gray level, below what the eye can distinguish on most displays, while at 4 bits they are 16 levels apart.

References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018 (Chapter 2, image sensing, sampling and quantization). publisher page
2. A. Torralba, P. Isola and W. T. Freeman, *Foundations of Computer Vision*, MIT Press, 2024 (open-access edition). book site
3. C. R. Harris et al., "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, 2020. doi
4. NumPy developers, "Data types," NumPy user guide, current documentation. docs
5. NumPy developers, "NEP 50: Promotion rules for Python scalars," NumPy Enhancement Proposals. NEP 50
6. S. van der Walt et al., "scikit-image: image processing in Python," *PeerJ*, vol. 2, e453, 2014. doi
7. scikit-image developers, "skimage.data: test images and example data," scikit-image documentation. docs

8. ITU-R, “Recommendation BT.601-7: Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios,” 2011. ITU page
9. Matplotlib developers, “matplotlib.pyplot.imshow,” Matplotlib documentation. docs
10. A. Dosovitskiy et al., “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale,” *International Conference on Learning Representations* (*ICLR*), 2021. arXiv:2010.11929
11. V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell and G. Wetzstein, “Implicit Neural Representations with Periodic Activation Functions,” *Advances in Neural Information Processing Systems* (*NeurIPS*), vol. 33, 2020. proceedings
12. A. Essakine et al., “Where Do We Stand with Implicit Neural Representations? A Technical and Performance Survey,” *Transactions on Machine Learning Research* (*TMLR*), 2025. arXiv:2411.03688
13. NumPy developers, “Indexing on ndarrays,” NumPy user guide, accessed 2026-11-05. documentation

Part II

Image Processing

IP 01 • Introduction

What digital image processing is, where it came from, which parts of the electromagnetic spectrum it works with, and the roadmap of fundamental steps this series follows.

Online edition: https://blog.shyandram.dev/en/learn/image-processing/introduction/

What you'll learn

- What a digital image is, in words and as a function $f(x, y)$, and what "processing" one means
- Where image processing stops and computer vision begins, using the low-, mid- and high-level view
- How the field started: newspaper photos sent by telegraph, lunar probes, and medical scanners
- Which parts of the electromagnetic spectrum (and which non-light signals) produce the images we process
- The fundamental steps of image processing, which double as the roadmap for the 13 chapters of this series
- The hardware and software components of a working image processing system, and how deep learning is reshaping the camera pipeline

The big picture

> **Explain like I'm five**
>
> Imagine a huge mosaic made of tiny square tiles. Each tile has one shade of gray, written as a number: 0 for black, 255 for white. A computer cannot "see" the mosaic. It can only read the list of numbers. **Digital image processing** is the set of tricks we use to change those numbers so the picture becomes clearer, smaller, or easier to understand: wiping away specks of dust, making a dark photo brighter, or drawing a line around every coin on a table so we can count them.

Almost every image you look at today has passed through image processing before it reached your eyes. Your phone ran dozens of operations between the moment light hit its sensor and the moment the photo appeared on screen. A hospital CT scanner never takes a "photo" at all: it measures how X-rays are absorbed along thousands of lines and *computes* the picture. A weather satellite records energy the human eye cannot see and turns it into a map you can read.

This chapter is the map before the journey. It defines the field, tells you where it came from, shows the range of signals it handles, and lays out the steps that the remaining chapters cover one by one [1]. Nothing here is hard, but every later chapter refers back to the vocabulary we build now.

What is digital image processing?

Plain version. A picture becomes *digital* when we chop it into a grid of tiny squares and store one number (or a few numbers, for color) per square. *Processing* means feeding that grid of numbers to a computer and getting something useful back.

Images as functions

Mathematically, a grayscale image is a two-dimensional function

$$f(x, y), \qquad (x, y) \in \mathbb{R}^2.$$

Here x and y are spatial coordinates, and the value $f(x, y)$ is the **intensity** (or *gray level*) of the image at that point. For a camera, intensity is proportional to the light energy that reached the sensor; for an X-ray film, to the radiation that passed through the body. In the real world x, y and f can all vary continuously.

An image is **digital** when all three quantities are finite and discrete. We sample the plane on an $M \times N$ grid and quantize each value to one of L levels:

$$f : \{0, 1, \ldots, M-1\} \times \{0, 1, \ldots, N-1\} \to \{0, 1, \ldots, L-1\}, \qquad L = 2^k.$$

M is the number of rows, N the number of columns, and k the number of bits per value ($k = 8$ gives the familiar range 0 to 255). Each grid element is a **pixel** (short for *picture element*). Chapter 2 covers how sampling and quantization are chosen; for now it is enough to know that an image is a matrix.

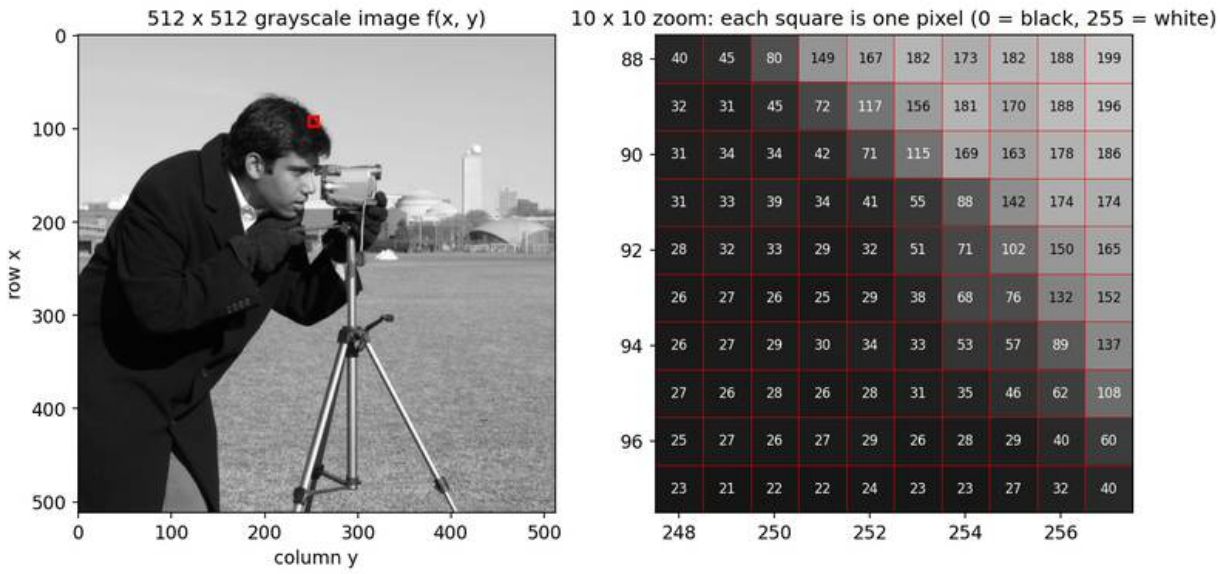


Figure 1.1 — Left: a 512 × 512, 8-bit grayscale image. Right: the 10 × 10 patch inside the red square, magnified. Each square is one pixel; the printed number is its gray level. The edge between dark hair and bright sky is nothing more than a jump from values near 30 to values near 180.

In code, this matrix is a NumPy array. Indexing with `[row, column]` reads a single pixel, and slicing returns a neighborhood:

```
from skimage import data

f = data.camera()            # a 512 x 512 grayscale photograph
print(f.shape, f.dtype)      # (512, 512) uint8
print(f[100, 250])           # intensity at row 100, column 250
print(f.min(), f.max())      # darkest and brightest values
patch = f[88:92, 248:252]    # a 4 x 4 neighborhood is just a sub-array
print(patch)
```

Which axis is x?

This series follows the textbook convention [1]: x indexes rows (top to bottom) and y indexes columns (left to right), so $f(x, y)$ matches NumPy's `f[x, y]`. Many graphics libraries, and the prerequisite tutorial Images as Arrays, use the opposite order (row $= y$, column $= x$). When code and math disagree, check the axes first.

Color images add a third index for the channel, $f(x, y, c)$, and medical volumes add a depth index, $f(x, y, z)$. The ideas in this chapter carry over unchanged.

Where processing ends and vision begins

People disagree about where *image processing* stops and *image analysis* or *computer vision* starts. One extreme says image processing covers only operations whose input and output are both images. That rule is clean but too narrow: it would exclude something as simple as computing the average brightness of a photo. The other extreme is computer vision, whose goal is to emulate human vision: to understand a scene, recognize objects and act on them. Textbooks on vision treat it as a field in its own right, with geometry, 3-D reconstruction and recognition at its core [7].

A more useful picture is a continuum with three kinds of processes [1]:

- **Low-level processes** take an image in and give an image out. Examples: removing noise, increasing contrast, sharpening. No interpretation is involved.
- **Mid-level processes** take an image in and give *attributes* out: regions, edges, contours, or measurements of individual objects. Segmentation (splitting an image into meaningful parts) and describing those parts in a form a computer can sort are typical mid-level tasks.
- **High-level processes** "make sense" of a collection of recognized objects. Reading a whole page of text, or deciding that a medical scan shows a tumor, are high-level tasks. This end of the continuum is where image analysis merges into computer vision.

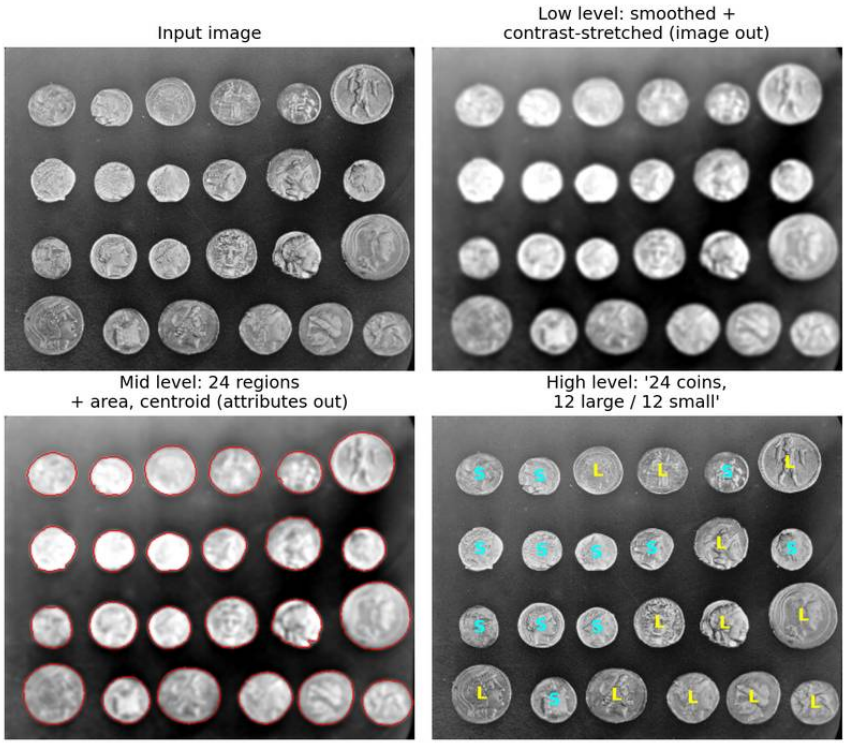


Figure 1.2 — The processing continuum on one image. Low level: denoise and stretch contrast (an image comes out). Mid level: segment the coins and measure each one (a list of regions and numbers comes out). High level: interpret the measurements (a statement comes out: 24 coins, half of them large).

In this series, "digital image processing" means everything from the low level up to and including the recognition of individual regions or objects. That is wide enough to include segmentation, feature extraction and classification (Chapters 10–13), but it stops short of full scene understanding.

The origins of digital image processing

Plain version. The first digital pictures were newspaper photos sent across the Atlantic as punched telegraph tape. Real computer processing of images began in the 1960s, when space agencies needed to clean up pictures from lunar probes, and it exploded in the 1970s with medical scanners that build pictures from measurements.

Pictures over a telegraph cable

In the early 1920s, the **Bartlane cable picture transmission system** sent newspaper photographs between London and New York over the transatlantic submarine telegraph cable [2]. A photograph was converted into codes punched on ordinary five-unit telegraph tape, sent with standard

telegraph equipment, and reassembled offline at the receiving end. According to Kobayashi's history of the system, it carried nearly 500 news pictures across the Atlantic until the outbreak of war in 1939 [2]. The earliest Bartlane pictures used only five distinct gray levels; by the end of the 1920s the number had grown to fifteen [1].

Strictly speaking, no *computer* touched these pictures, so this is the prehistory of digital image processing. But the core idea was already there: a picture is turned into a finite set of codes, transmitted, and rebuilt, and the number of gray levels determines how faithful the result is.



Figure 1.3 — How much do gray levels matter? The same photo quantized to 5 levels (early Bartlane), 15 levels (late 1920s Bartlane) and 256 levels (modern 8-bit). With 5 levels, smooth areas such as the coat and the grass collapse into flat patches; even with 15 levels the sky still shows visible bands. This artifact, called false contouring, is explained in Chapter 2.

You can reproduce this experiment in a few lines. Uniform quantization to L levels maps an intensity $r \in [0, 1]$ to

$$Q_L(r) = \frac{\operatorname{round}\left(r\,(L-1)\right)}{L-1},$$

where round rounds to the nearest integer, so the output takes only the L values $0, \frac{1}{L-1}, \ldots, 1$.

```
import numpy as np
from skimage import data

def quantize(img, levels):
    """Map a uint8 image onto `levels` evenly spaced gray levels."""
    x = img.astype(np.float64) / 255.0
    q = np.round(x * (levels - 1)) / (levels - 1)
    return (q * 255).astype(np.uint8)

f = data.camera()
for L in (5, 15, 256):
    g = quantize(f, L)
    print(L, "levels ->", len(np.unique(g)), "distinct values,",
          f"mean abs error {np.abs(g.astype(int) - f).mean():.2f}")
```

On the test image, the mean absolute error falls from about 18 gray levels with $L = 5$ to about 5 with $L = 15$, and to 0 with $L = 256$.

Real computer processing of pictures began in the 1960s with the space program: computers were used to correct the distortions in images sent back by lunar probes, one of the founding projects of digital image processing [1].

Online only: Computers meet images: the space program — https://blog.shyandram.dev/en/learn/image-processing/introduction/notes/computers-meet-images-the-space-program/

Images computed from measurements: CT

The second founding application came from medicine. In 1973, Godfrey Hounsfield described a system for **computerized transverse axial scanning**, now called computed tomography (CT) [4]. Instead of exposing one film, an X-ray source and detectors rotate around the patient and record how strongly the beam is absorbed along many lines at many angles. A computer then calculates the absorption at every point of a cross-section. Hounsfield reported that this revealed soft-tissue differences that conventional X-ray films could not show [4].

CT is a milestone because the image is not *captured*; it is *reconstructed*. The scanner measures line integrals of the unknown slice $f(x, y)$:

$$p(s, \theta) = \iint_{\mathbb{R}^2} f(x, y)\, \delta(x \cos\theta + y \sin\theta - s)\, dx\, dy,$$

where θ is the angle of the beam, s is the detector position along the projection, and δ is the Dirac delta, which keeps only the points on the line $x \cos\theta + y \sin\theta = s$. The collection of all projections $p(s, \theta)$ is the **sinogram**. Recovering f from p is an inverse problem, the subject of the reconstruction half of Chapter 5. The standard test object for such algorithms is a synthetic head slice made of ellipses, introduced by Shepp and Logan in 1974 [5].

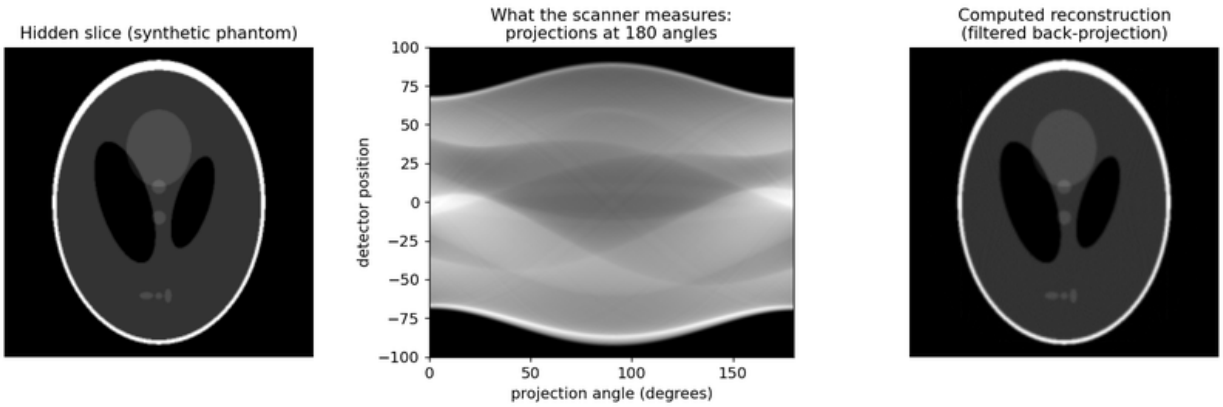


Figure 1.4 — Computed tomography in miniature. Left: an unknown slice (the Shepp–Logan phantom [5]). Middle: what the scanner actually records, one projection per angle. Right: the slice recovered by filtered back-projection, an algorithm derived in Chapter 5.

```
import numpy as np
from skimage.data import shepp_logan_phantom
from skimage.transform import radon, iradon, rescale

slice_ = rescale(shepp_logan_phantom(), 0.5)          # 200 x 200 synthetic head slice
angles = np.linspace(0, 180, 180, endpoint=False)
sinogram = radon(slice_, theta=angles)                 # what the scanner records
recon = iradon(sinogram, theta=angles, filter_name="ramp")
err = np.sqrt(np.mean((recon - slice_) ** 2))
print(sinogram.shape, recon.shape, f"RMS error {err:.3f}")   # (200, 180) (200, 200) ~0.025
```

From the 1960s onward, the same ideas spread into remote sensing, astronomy, biology, industrial inspection, law enforcement and, eventually, every phone in every pocket. Two broad goals drive all of these applications: improving pictures for *human* interpretation, and processing image data for storage, transmission and *machine* perception.

Examples of fields that use digital image processing

Plain version. Our eyes see only a thin slice of the "light" that exists. Other kinds of light, from gamma rays to radio waves, can also make pictures, and so can sound and electron beams. Image processing works on all of them, because they all end up as grids of numbers.

The most useful way to organize applications is by the **source** of the image [1]. The main source is electromagnetic (EM) energy; others are acoustic waves, electrons, and pure computation.

The electromagnetic spectrum

EM waves can be described by wavelength λ or frequency ν, which are linked by

$$\lambda = \frac{c}{\nu},$$

where $c \approx 2.998 \times 10^8$ m/s is the speed of light. Light also comes in packets, **photons**, each carrying energy

$$E = h\nu = \frac{hc}{\lambda},$$

where $h \approx 6.626 \times 10^{-34}$ J·s is Planck's constant. Short wavelength means high frequency and energetic photons; long wavelength means low frequency and gentle photons. That single fact explains much of imaging: energetic photons pass through tissue and metal (X-rays), while low-energy waves pass through clouds and darkness (radar and radio).

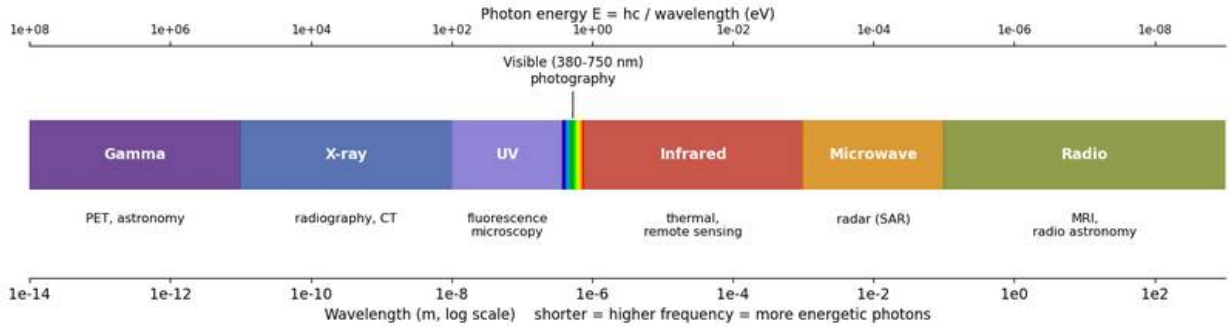


Figure 1.5 — The electromagnetic spectrum on a log scale, with photon energy on the top axis. The visible band is a sliver. Band boundaries are conventions and vary between sources.

```
h, c, eV = 6.626e-34, 2.998e8, 1.602e-19   # Planck, speed of light, joules per eV

def photon_energy_ev(wavelength_m):
    return h * c / wavelength_m / eV

for name, lam in [("X-ray (0.1 nm)", 1e-10), ("green light (550 nm)", 550e-9),
                  ("thermal IR (10 um)", 10e-6), ("radar (3 cm)", 0.03)]:
    print(f"{name:22s} {photon_energy_ev(lam):10.3g} eV")
# X-ray ~1.24e4 eV, green ~2.25 eV, thermal IR ~0.124 eV, radar ~4.1e-5 eV
```

An X-ray photon carries roughly ten thousand times more energy than a photon of green light, and a radar photon carries less than a ten-thousandth of it. The bands below run from the most energetic to the least.

From gamma rays to radio waves

Plain version. Each band is good at showing something different, and each brings its own kind of noise. A quick tour from the most energetic photons to the least:

- **Gamma rays:** nuclear medicine, where the image shows where a radioactive tracer collects (function rather than anatomy), and astronomy. The images are noisy because each pixel counts only a modest number of photons.
- **X-rays:** radiographs, angiography and CT [4] in medicine; hidden defects in welds and circuit boards in industry.
- **Ultraviolet:** fluorescence microscopy, where tagged structures glow against a dark background.
- **Visible and infrared:** photography, microscopy, machine vision and multispectral remote sensing; thermal infrared senses emitted heat rather than reflected light.
- **Microwaves:** imaging radar, which carries its own illumination and sees through clouds and darkness [8].
- **Radio waves:** MRI [6] in medicine, and radio astronomy, where the first image of a black hole's shadow was reconstructed from the data of many telescopes [9].

The one result to remember: CT, MRI, radar and radio astronomy do not capture a picture directly; they compute it from measurements.

Online only: A band-by-band tour of the spectrum — https://blog.shyandram.dev/en/learn/image-processing/introduction/notes/a-band-by-band-tour-of-the-spectrum/

Other imaging modalities

Not every image comes from EM radiation.

- **Acoustic imaging.** Sound waves reflect at boundaries between materials. Geologists send low-frequency sound into the ground and record echoes to map underground rock layers for oil, gas and mineral exploration (*seismic imaging*). Medical **ultrasound** uses high-frequency sound (millions of cycles per second) to image the fetus, the heart and abdominal organs in real time and without ionizing radiation. In both cases the image is formed by timing the echoes.
- **Electron microscopy.** Electrons can be focused like light, but their effective wavelength is far shorter, so electron microscopes resolve much finer detail than light microscopes. A *transmission* electron microscope passes a beam through a thin specimen; a *scanning* electron microscope sweeps a focused beam across a surface and records the electrons that bounce off or are knocked out. The images are often noisy and benefit from enhancement and restoration.
- **Synthetic images.** Computers also generate images that no sensor captured: fractals, rendered 3-D scenes, simulations, and, increasingly, images produced by generative models. These images are processed with the same tools, and they are widely used to create training data and test cases.

The lesson of this tour: once the data is a grid of numbers, the same toolbox applies, whether the numbers came from gamma rays, sound echoes or a computer program. What changes is the noise, the resolution and the physics of formation, and those differences decide which tool you reach for.

Fundamental steps in digital image processing

Plain version. Making an image useful is like cooking: you get the ingredients (acquire the image), clean and prepare them (enhance, restore), sometimes change their form (color, transforms, compression), then cut them into pieces (segmentation), describe each piece, and finally decide what each piece is (classification). You do not always need every step.

It helps to split the steps into two groups [1]: methods whose outputs are generally **images**, and methods whose outputs are generally **attributes** extracted from images. A **knowledge base** about the problem domain, such as where defects tend to appear or what a valid label looks like, guides every step. The list is not a mandatory pipeline; a given application uses only the steps it needs, in whatever order makes sense.

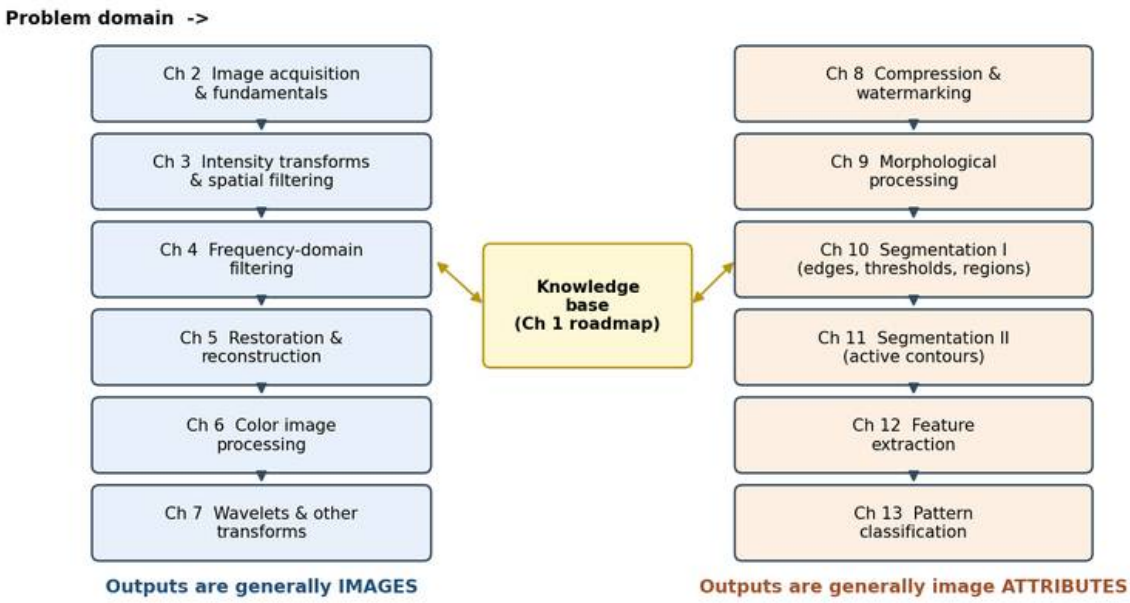


Figure 1.6 — The fundamental steps of digital image processing, labeled with the chapter of this series that covers each. Steps on the left generally produce images; steps on the right generally produce attributes. Domain knowledge (center) informs all of them.

Here is each step, with the chapter that develops it.

1. **Image acquisition** — obtaining the image in digital form, including sensing, sampling and quantization. Chapter 2 also builds the mathematical toolkit (pixel relationships, arithmetic, geometric transforms) used everywhere else. → IP 02 ·Digital Image Fundamentals
2. **Image enhancement** — making an image more suitable for a specific purpose, such as increasing contrast or sharpening detail. Enhancement is subjective: "better" depends on the viewer and the task. → IP 03 ·Intensity Transformations and Spatial Filtering and IP 04 ·Filtering in the Frequency Domain
3. **Image restoration** — undoing a known or estimated degradation such as blur or noise. Unlike enhancement, restoration is objective: it relies on mathematical models of how the image was degraded. The same chapter covers reconstruction from projections (CT). → IP 05 ·Image Restoration and Reconstruction
4. **Color image processing** — color models, pseudo-color, and processing of full-color images. → IP 06 ·Color Image Processing
5. **Wavelets and other image transforms** — representing images at several resolutions at once, the basis of modern compression and pyramid methods. → IP 07 ·Wavelet and Other Image Transforms
6. **Compression and watermarking** — reducing the storage or bandwidth an image needs, and embedding invisible information to protect ownership. → IP 08 ·Image Compression and Watermarking
7. **Morphological processing** — tools for extracting and cleaning up shape components, such as thinning, filling holes and separating touching objects. This is where outputs start to become

attributes. → IP 09 ·Morphological Image Processing

8. **Segmentation** — partitioning an image into its constituent parts or objects. It is one of the hardest steps, and errors here propagate to everything after it. → IP 10 ·Image Segmentation I (edges, thresholds, regions) and IP 11 ·Image Segmentation II (active contours: snakes and level sets)
9. **Feature extraction** — turning segmented regions into numbers: boundary and region descriptors, and point features such as corners. It has two parts: *detecting* features and *describing* them. → IP 12 ·Feature Extraction
10. **Image pattern classification** — assigning a label to an object based on its features, from minimum-distance classifiers to neural networks and deep learning. → IP 13 ·Image Pattern Classification

Chapter 1 (this page) is the overview that ties them together.

The following sketch strings four of these steps together on the coins image of Figure 1.2. Every step is one or two library calls here; the chapters explain what is going on inside each.

```
from scipy import ndimage as ndi
from skimage import data, filters, measure, segmentation

img = data.coins()

# 1) enhancement: low level, image in -> image out
smooth = filters.gaussian(img, sigma=1, preserve_range=True)

# 2) segmentation: mid level, image in -> regions out
markers = (smooth < 30) * 1 + (smooth > 150) * 2       # sure background / sure coin
regions = segmentation.watershed(filters.sobel(smooth), markers) == 2
regions = ndi.binary_fill_holes(regions)
labels = measure.label(regions)

# 3) feature extraction: regions in -> numbers out
props = [p for p in measure.regionprops(labels) if p.area > 300]
areas = [p.area for p in props]

# 4) classification / interpretation: numbers in -> a statement out
median = sorted(areas)[len(areas) // 2]
n_large = sum(a >= median for a in areas)
print(f"{len(areas)} coins found, {n_large} at or above the median size")
# 24 coins found, 12 at or above the median size
```

Notice the knowledge base hiding in this code: the thresholds 30 and 150, the minimum area 300 and the rule "large means at or above the median" all encode facts about this particular problem. Change the lighting or the coins and those numbers must change too. Much of the history of the field, including the rise of deep learning, is the story of moving such knowledge from hand-set constants into models learned from data.

Components of an image processing system

Plain version. An image processing system is a team: a sensor that catches the image, fast special-purpose chips that do the heavy lifting, a computer that runs the programs, memory that stores the pictures, a screen to look at them, a printer for paper copies, and a network that connects it all.

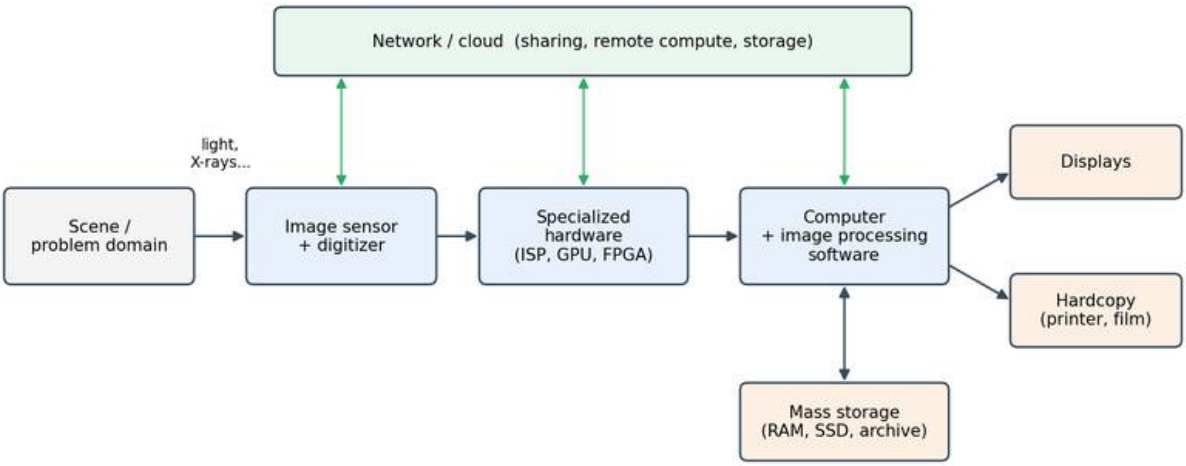


Figure 1.7 — Components of a general-purpose image processing system. Arrows show the main data flow; the network connects every stage to remote storage and computation.

- **Image sensors.** Two elements are needed to acquire an image: a physical device that responds to the energy radiated by the object (a CCD or CMOS chip, an X-ray detector, an ultrasound transducer), and a **digitizer** that converts the device's electrical output into numbers. In a phone, both sit on the same chip.
- **Specialized image processing hardware.** Some operations must run at video rate on every pixel, faster than a general-purpose processor can manage. Classic systems used dedicated arithmetic boards; today the role is filled by the *image signal processor* (ISP) inside every camera chip, by graphics processing units (GPUs), by field-programmable gate arrays (FPGAs), and by neural processing units in phones.
- **The computer.** Anything from an embedded microcontroller in a smart doorbell to a cluster of servers. For dedicated tasks, the computer may be customized; for research and general work, an ordinary workstation is enough.
- **Software.** Modules that perform specific tasks, plus a way to combine them. In this series we use Python with NumPy, SciPy, scikit-image and OpenCV. The ability to write your own code, rather than only call existing functions, is what this series aims to build.
- **Mass storage.** Images are large, and they arrive in bulk. Storage comes in three tiers: *short-term* storage during processing (RAM, GPU memory, frame buffers), *online* storage for fast recall (SSDs and disk arrays), and *archival* storage for rare access (tape, cold cloud storage).
- **Image displays.** Mostly color flat panels today. For medical diagnosis, specialized calibrated

monitors are used, and for stereo or virtual reality, head-mounted displays.
- **Hardcopy devices.** Laser and inkjet printers, film cameras, and heat-sensitive printers. Film still offers very high resolution, and paper remains the medium of choice for written material and many reports.
- **Networking and the cloud.** Practically every system is now connected. Transmitting images is a bandwidth problem, which is why compression (Chapter 8) is a core topic. Cloud services move both storage and heavy computation (for example, training deep models) off the device.

How large is "large"? An image with M rows, N columns, C channels and k bits per value needs

$$b = M \times N \times C \times k \text{ bits}$$

before compression, where $C = 1$ for grayscale and $C = 3$ for RGB color.

```
def raw_size_mb(height, width, channels=1, bits=8, frames=1):
    return height * width * channels * bits * frames / 8 / 1e6

print(f"{raw_size_mb(512, 512):.2f} MB   - this chapter's 8-bit test photo")         # 0.26
print(f"{raw_size_mb(3000, 4000, 3):.0f} MB   - one 12-megapixel RGB photo")         # 36
print(f"{raw_size_mb(512, 512, bits=16, frames=300):.0f} MB  - a 300-slice 16-bit CT volume")
 ↪  # 157
print(f"{raw_size_mb(2160, 3840, 3, frames=30*60):.0f} MB - one minute of uncompressed 4K
 ↪  video")  # 44790
```

One minute of uncompressed 4K video at 30 frames per second is about 45 gigabytes. Without compression, streaming video and image-heavy web pages would be impractical.

Modern view

The textbook's view of the field, a pipeline of well-understood, hand-designed steps, is still the right way to *learn* image processing. Two developments since about 2010 have changed how the field is *practiced*: computation has moved into the image formation itself, and learned models have replaced or augmented many hand-designed steps. The surveys below are good entry points.

Surveys on computational imaging and imaging beyond the visible

- **Mait, Euliss and Athale, "Computational imaging" (2018)** [8] review two decades of systems in which optics and computation are designed together, so that the sensor records data that is not a recognizable picture until it is processed. Their survey spans topics such as spatial light modulators, synthetic aperture radar and computational microscopy. The main takeaway is that the boundary between "acquisition" and "processing" in Figure 1.6 is now a design choice: CT, MRI and radar were early examples, and phones have made the approach mainstream.

- **Delbracio, Kelly, Brown and Milanfar, "Mobile computational photography: a tour" (2021)** [10] trace how phone cameras overcame their tiny sensors and lenses with algorithms: capturing a burst of frames and merging them, aggressive noise reduction, and multi-frame super-resolution. Their tour shows that a modern phone photo is the output of a long processing pipeline, and it is a useful companion to almost every chapter in this series.
- **Gallego et al., "Event-based vision: a survey" (2020)** [11] cover a sensor that breaks the frame-based model of Figure 1.1 entirely. An *event camera* pixel fires asynchronously whenever its own brightness changes, giving microsecond timing and very high dynamic range. The survey shows how classic tasks such as feature detection, optical flow and image reconstruction must be rethought when the input is a stream of events rather than a grid of numbers.
- **Wang, Ye and De Man, "Deep learning for tomographic image reconstruction" (2020)** [12] review how neural networks entered CT, MRI, PET and related modalities, the reconstruction problems that launched the field in the 1970s. They cover networks that clean up conventional reconstructions, physics-informed approaches that keep the measurement model in the loop, direct learned reconstruction, and generative models, with goals such as faster scans, lower radiation dose and better image quality. Their outlook stresses that large, well-curated datasets will be central to further progress.

How deep learning reshaped the processing pipeline

The broad shift is summarized in the review by **LeCun, Bengio and Hinton (2015)** [13]: deep networks learn representations of data at several levels of abstraction directly from examples, instead of relying on features designed by hand. For image processing, this has played out differently at each level of the continuum in Figure 1.2.

At the **high level**, the change has been nearly complete. Classification and recognition (Chapter 13) are now dominated by learned models; Chapters 12 and 13 explain the hand-designed features and classifiers that came before, which remain useful when data is scarce or interpretability matters.

At the **low level**, the clearest case study is the camera's own processing pipeline. A conventional ISP is a chain of separately tuned stages that turn raw sensor values into a finished photo: black-level correction, demosaicing, denoising, white balance, color correction, tone mapping and compression. **Ramanath et al. (2005)** [14] give a clear overview of this pipeline and its trade-offs, and it is a good reference to keep next to Chapters 3, 5 and 6.

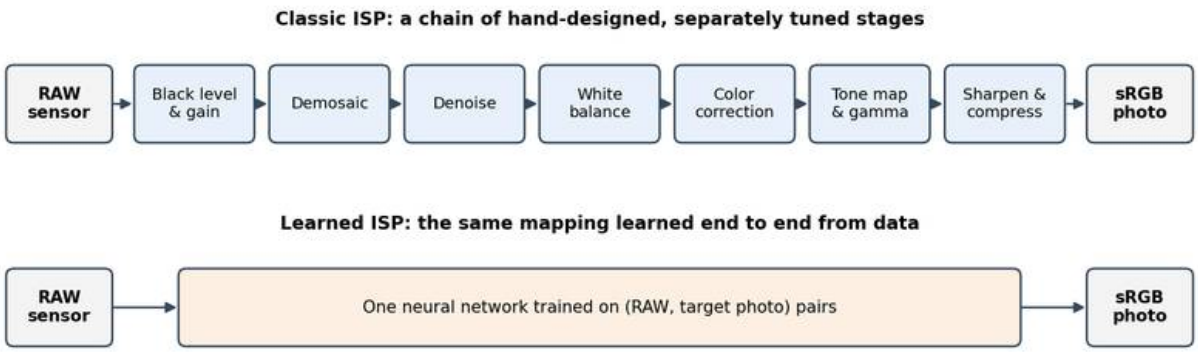


Figure 1.8 — A classic camera ISP (top) is a chain of hand-designed stages, each covered somewhere in this series. A learned ISP (bottom) replaces the chain, or parts of it, with one network trained on pairs of raw data and target photos.

Deep learning has attacked this pipeline from several directions:

- **Learning a hard stage end to end.** Chen et al. (2018) trained a fully convolutional network to map short-exposure raw images taken in very low light directly to clean photos, using long-exposure shots of the same scenes as targets [15]. A traditional pipeline amplifies noise badly in this regime.
- **Making training data realistic.** Brooks et al. (2018) showed how to "unprocess" ordinary photos, inverting the steps of the camera pipeline to synthesize realistic raw data, so that denoising networks trained on it work on real sensor output [16]. This is a good example of classic pipeline knowledge making a learned model better.
- **Replacing the whole ISP.** Ignatov, Van Gool and Timofte (2020) proposed PyNET, a pyramid-shaped convolutional network that maps raw data from a smartphone sensor directly to a high-quality photo, learning demosaicing, denoising, color and tone in a single model [17]. They trained it on paired raw phone images and photos from a high-end DSLR camera of the same scenes.
- **Surveying the space.** da Silva et al. (2023) survey deep learning methods for image signal processing, comparing approaches that replace individual stages with those that replace the entire ISP with one network [18].

What did **not** change? The mathematics of image formation in Chapters 2, 4 and 5 still determines what any method, learned or not, can recover. Learned models are trained and evaluated with the quality measures, transforms and color spaces defined in the classic chapters. And many deployed systems are hybrids: a learned denoiser inside an otherwise conventional pipeline, or a physics-based reconstruction refined by a network [12]. Understanding the classic steps is what lets you see which part of such a system is doing what, and why it fails when it does.

Key takeaways

- A digital image is a function $f(x, y)$ sampled on an $M \times N$ grid and quantized to $L = 2^k$ levels; in code it is simply an array.
- Image processing spans a continuum: low-level (image to image), mid-level (image to attributes) and high-level (attributes to meaning). Computer vision lives at the high end.
- The field grew from telegraph picture transmission (1920s), computer correction of space-probe images (1960s) and computed medical images such as CT (1970s).
- Images come from every part of the EM spectrum, from gamma rays to radio waves, and from non-EM sources such as ultrasound, seismic waves and electron beams. The physics of the source determines noise and resolution, and so which tools you need.
- The fundamental steps split into steps whose outputs are images (acquisition through compression) and steps whose outputs are attributes (morphology through classification), all guided by domain knowledge. Chapters 2–13 follow this roadmap.
- A working system combines sensors, specialized hardware, a computer, software, storage, displays, hardcopy and networking.
- Deep learning has largely taken over high-level recognition and is reshaping low-level steps such as the camera ISP, but the classic models of image formation remain the foundation that learned methods build on.

Exercises

1. **Storage budget.** A microscope records a time-lapse of 16-bit grayscale images of size 2048×2048, one every 30 seconds for 48 hours. How much uncompressed storage does the experiment need? How many frames would fit on a 1 TB disk?

   Hint

   Each frame is $2048 \times 2048 \times 16$ bits $= 8{,}388{,}608$ bytes, about 8.4 MB. There are $48 \times 120 = 5760$ frames, so about 48 GB in total. A 1 TB disk holds roughly $10^{12}/8.39 \times 10^6 \approx 119{,}000$ frames.

2. **Place it on the continuum.** Classify each task as low-, mid- or high-level, and justify your answer: (a) removing salt-and-pepper noise from a scanned page; (b) outlining every cell in a microscope image and reporting each cell's area; (c) deciding whether a chest X-ray is normal; (d) brightening a dark photo.

   Hint

   Ask what comes *out*. An image out means low level ((a), (d)). Regions or measurements out means mid level ((b)). A judgment about the scene means high level ((c)).

3. **Why X-rays and not light?** Using $E = hc/\lambda$, compute the photon energy of a 0.05 nm X-ray and of 600 nm orange light. Explain in two sentences why the first is used to image bones and the second is not.

Hint

About 24.8 keV versus about 2.07 eV, a ratio of roughly 12,000. Visible photons are absorbed or scattered within millimeters of tissue; high-energy X-ray photons pass through soft tissue, and bones absorb a larger share of them, which produces contrast.

4. **How many gray levels?** Using the `quantize` function from this chapter, plot the mean absolute error between `data.camera()` and its quantized version for $L = 2, 3, \ldots, 32$. Then look at the images. Around which L do false contours stop being obvious to you? Is the error curve a good predictor of what you see?

Hint

For uniform quantization the error falls roughly as $1/L$. Visible banding depends on the smooth areas of the image (here, the sky) more than on the average error, which is one reason Chapter 2 treats perceived quality separately from numerical error.

5. **Design a system.** A factory wants to reject cracked ceramic tiles moving on a conveyor belt at two tiles per second. List the components of Figure 1.7 you would need, and for each of the fundamental steps of Figure 1.6, say whether you would use it and why.

Hint

You need a sensor with controlled lighting, enough compute to process two images per second, and a link to the reject mechanism. Likely steps: acquisition, enhancement (contrast), perhaps morphology, segmentation of crack-like structures, features (crack length), and a classification rule. Color, wavelets and compression are probably unnecessary, unless images are archived for audit.

6. **Learned or classic?** Pick one stage of the classic ISP in Figure 1.8. Give one argument for replacing it with a learned model and one argument for keeping the hand-designed version.

Hint

For learning: a network can adapt to the actual sensor noise and scene statistics. For keeping it: a hand-designed stage is predictable, cheap to run, easy to tune and does not need paired training data. Several references in the Modern view discuss exactly this trade-off.

References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018, Ch. 1. publisher page
2. K. Kobayashi, "Birth of a Digital Phototelegraph — the Bartlane System," *Journal of the Institute of Image Electronics Engineers of Japan*, vol. 31, no. 2, pp. 244–249, 2002. J-STAGE
3. NASA, "First Image of the Moon Taken by a U.S. Spacecraft" (Ranger 7, PIA02975), NASA Photojournal. NASA

4. G. N. Hounsfield, “Computerized transverse axial scanning (tomography): Part 1. Description of system,” *British Journal of Radiology*, vol. 46, no. 552, pp. 1016–1022, 1973. doi:10.1259/0007-1285-46-552-1016
5. L. A. Shepp and B. F. Logan, “The Fourier reconstruction of a head section,” *IEEE Transactions on Nuclear Science*, vol. 21, no. 3, pp. 21–43, 1974. doi:10.1109/TNS.1974.6499235
6. P. C. Lauterbur, “Image formation by induced local interactions: examples employing nuclear magnetic resonance,” *Nature*, vol. 242, pp. 190–191, 1973. doi:10.1038/242190a0
7. R. Szeliski, *Computer Vision: Algorithms and Applications*, 2nd ed., Springer, 2022. book site
8. J. N. Mait, G. W. Euliss and R. A. Athale, “Computational imaging,” *Advances in Optics and Photonics*, vol. 10, no. 2, pp. 409–483, 2018. doi:10.1364/AOP.10.000409
9. The Event Horizon Telescope Collaboration, “First M87 Event Horizon Telescope Results. I. The Shadow of the Supermassive Black Hole,” *The Astrophysical Journal Letters*, vol. 875, no. 1, 2019. doi:10.3847/2041-8213/ab0ec7
10. M. Delbracio, D. Kelly, M. S. Brown and P. Milanfar, “Mobile Computational Photography: A Tour,” arXiv:2102.09000, 2021. arXiv
11. G. Gallego et al., “Event-based Vision: A Survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020. arXiv
12. G. Wang, J. C. Ye and B. De Man, “Deep learning for tomographic image reconstruction,” *Nature Machine Intelligence*, vol. 2, no. 12, pp. 737–748, 2020. doi:10.1038/s42256-020-00273-z
13. Y. LeCun, Y. Bengio and G. Hinton, “Deep learning,” *Nature*, vol. 521, pp. 436–444, 2015. doi:10.1038/nature14539
14. R. Ramanath, W. E. Snyder, Y. Yoo and M. S. Drew, “Color image processing pipeline,” *IEEE Signal Processing Magazine*, vol. 22, no. 1, pp. 34–43, 2005. doi:10.1109/MSP.2005.1407713
15. C. Chen, Q. Chen, J. Xu and V. Koltun, “Learning to See in the Dark,” *CVPR*, 2018. arXiv
16. T. Brooks, B. Mildenhall, T. Xue, J. Chen, D. Sharlet and J. T. Barron, “Unprocessing Images for Learned Raw Denoising,” arXiv:1811.11127, 2018. arXiv
17. A. Ignatov, L. Van Gool and R. Timofte, “Replacing Mobile Camera ISP with a Single Deep Learning Model,” arXiv:2002.05509, 2020. arXiv
18. M. H. M. da Silva et al., “ISP meets Deep Learning: A Survey on Deep Learning Methods for Image Signal Processing,” arXiv:2305.11994, 2023. arXiv

IP 02 • Digital Image Fundamentals

How eyes and cameras turn light into numbers, how sampling and quantization shape a digital image, and the basic math every later chapter builds on.

Online edition: https://blog.shyandram.dev/en/learn/image-processing/fundamentals/

What you'll learn

- How the human eye senses light, and why what we *see* is not always what is *there* (brightness adaptation, the Weber ratio, Mach bands, simultaneous contrast)
- Where images come from physically: the electromagnetic spectrum, sensors, and the illumination–reflectance model $f = i \cdot r$
- What sampling and quantization do, how they set spatial and intensity resolution, and how nearest, bilinear and bicubic interpolation resize an image
- The vocabulary of pixel relationships: neighbors, adjacency, connectivity, regions, boundaries, and three distance measures
- The mathematical toolbox used in every later chapter: elementwise vs matrix operations, linearity, arithmetic, set and logical operations, geometric transforms, image transforms, and histograms as probabilities

The big picture

Chapter 1 told us *what* digital image processing is. This chapter explains *what a digital image actually is*, starting from the light that leaves a scene and ending with a grid of integers in memory. Along the way it defines the words — pixel, neighbor, resolution, linear operator — that the rest of the series uses without stopping to explain.

> **Explain like I'm five**
>
> Imagine copying a painting onto graph paper with a box of crayons. First you decide how many squares the graph paper has: more squares means you can draw smaller details. That is **sampling**. Then you decide how many crayon colors you are allowed to use: more crayons means smoother shading. That is **quantization**. A digital image is just that graph paper, with a number written in each square saying which crayon you picked. Everything else in image processing is a recipe for changing those numbers.

Two things make this chapter more than bookkeeping. First, the final consumer of most images is a human, so we need to know where the eye is easily fooled and where it is sharp. Second, every choice made when an image is captured — how many samples, how many levels, which interpolation — leaves a fingerprint that later algorithms must live with. Deep networks did

not remove these choices; they mostly moved them into training data and loss functions, as the Modern view section shows.

Elements of visual perception

In plain words: your eye is a camera made of jelly and nerve cells. It is extremely good at seeing *differences* and *edges*, and surprisingly bad at judging absolute brightness.

Structure of the eye and image formation

Light enters through the cornea, passes the pupil (whose size the iris controls), and is focused by the lens onto the **retina**, a thin layer of light receptors at the back of the eye [1]. The lens is flexible: ciliary muscles change its curvature, which changes its focal length so that near and far objects can both be brought into focus. The retinal image is real and inverted, just like the image on a camera sensor.

The retina contains two kinds of receptor [1]:

- **Cones** — a few million per eye, packed most densely in a small central pit called the **fovea**. They work in bright light (*photopic* vision), come in three types with different spectral sensitivities, and give us color and fine detail. Many cones have their own nerve connection, which is why foveal vision is sharp.
- **Rods** — many times more numerous, spread across the retina, absent from the center of the fovea. They work in dim light (*scotopic* vision), have no color discrimination, and many rods share one nerve ending, trading detail for sensitivity. This is why objects seen by moonlight look colorless and soft.

The point where the optic nerve leaves the eye has no receptors at all: the **blind spot**. You do not notice it because the brain fills in the gap — an early hint that perception is an active reconstruction, not a recording.

Brightness adaptation and discrimination

The visual system copes with an enormous range of light levels, from starlight to sunlight. It does this by **brightness adaptation**: at any moment it operates over a much narrower window centered on the current adaptation level, and that window slides as the overall light changes [1]. Within the window, subjective brightness grows roughly with the *logarithm* of the physical light intensity, not linearly.

How small a change can we notice? Put a uniform field of intensity I in front of an observer and flash a small spot of intensity $I + \Delta I$. The increment that is noticed half of the time is ΔI_c. The ratio

$$W = \frac{\Delta I_c}{I}$$

is the **Weber ratio**, where I is the background intensity and ΔI_c is the just-noticeable increment. A *small* W means good discrimination (a small relative change is visible). Over a wide middle range of intensities W is roughly constant, which is Weber's law: what we perceive is the *relative* change, not the absolute one. At low intensities, where rods dominate, W is large and discrimination is poor [1].

This has a direct engineering consequence. Because we perceive relative changes, equally spaced *physical* gray levels are not equally spaced *perceptually*. That is one reason images are stored with a nonlinear (gamma) encoding, and one reason 8 bits are usually enough for display.

Mach bands, simultaneous contrast and illusions

Perceived brightness is not a simple function of intensity. Two classic demonstrations make this obvious [1]:

- **Mach bands.** Show a staircase of flat gray stripes. Each stripe is physically uniform, yet observers report a thin darker band on the dark side of every edge and a thin brighter band on the bright side. The visual system exaggerates edges, a behavior consistent with lateral inhibition between neighboring receptive fields — essentially a built-in sharpening filter.
- **Simultaneous contrast.** A mid-gray square looks lighter on a dark background and darker on a light one, although the square is identical in both cases. Perceived brightness depends on the surround.

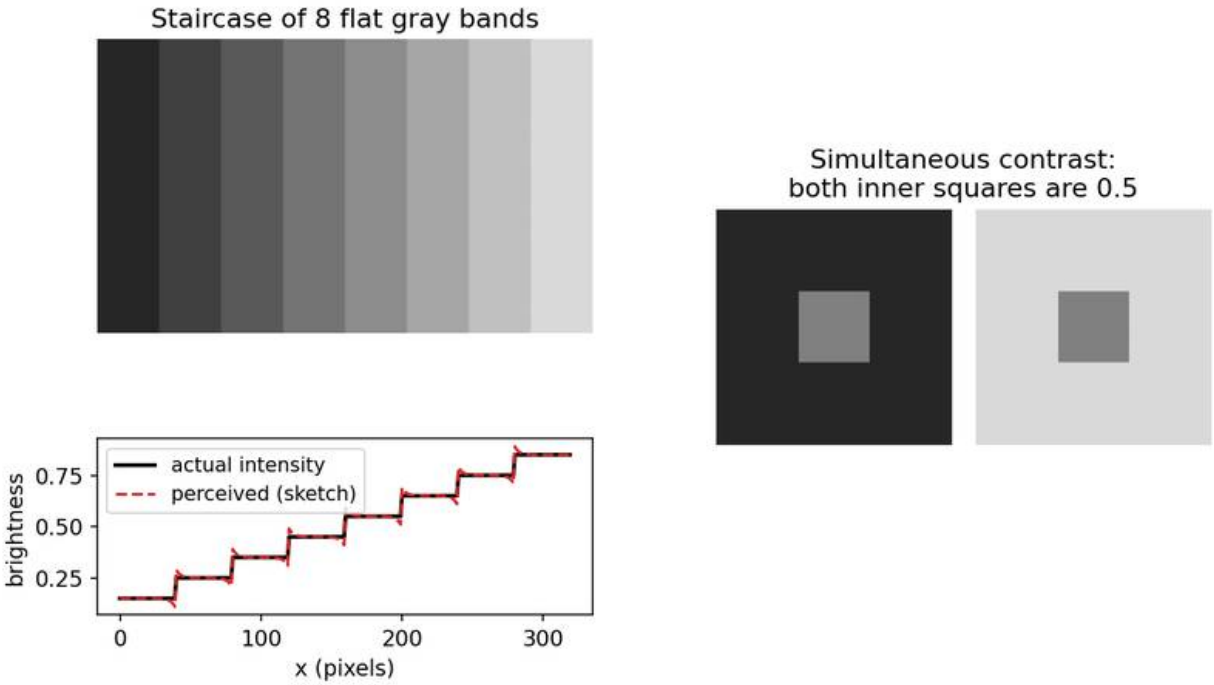


Figure 2.1 — Left: a staircase of uniform bands; the dashed curve sketches the over- and undershoot observers report at each edge (Mach bands). Right: simultaneous contrast — both inner squares have intensity 0.5.

Optical illusions in general — contours that are not drawn, lines of equal length that look unequal — show that the eye fills in and reinterprets. For image processing this matters twice: an algorithm that minimizes squared error may still produce images that *look* wrong, and an artifact that is numerically small may be very visible if it sits on an edge. This is the starting point of perceptual quality metrics, discussed in the Modern view.

Light and the electromagnetic spectrum

In plain words: light is a wave of energy. Visible light is a thin slice of a much wider spectrum that also includes radio, microwaves, infrared, ultraviolet, X-rays and gamma rays — and images can be made from any of them.

Wavelength, frequency and energy

Electromagnetic radiation can be described as a sinusoidal wave or as a stream of massless particles called photons [1]. The two descriptions are linked by

$$\lambda = \frac{c}{\nu}, \qquad E = h\nu,$$

where λ is the wavelength (meters), ν is the frequency (hertz), $c \approx 3 \times 10^8$ m/s is the speed of light, E is the energy of one photon (joules), and h is Planck's constant. Shorter wavelength means higher frequency and higher energy per photon. That is why gamma rays and X-rays can pass through tissue, while radio waves need large antennas to be detected.

The visible band spans roughly 400 nm (violet) to 700 nm (red). An object's perceived color is determined by which wavelengths it reflects: a green leaf reflects mostly in the middle of the band and absorbs the rest. Light with no color, only intensity, is called **achromatic** or **monochromatic**, and its intensity is called the **gray level**.

Measuring light: three terms

"How much light" can mean three different things. **Radiance** is the physical energy a source emits, **luminance** is the part of it the eye can perceive, and **brightness** is the subjective sensation, which depends on adaptation and surroundings. The point to remember: a sensor, like the eye, only sees energy in the band it is sensitive to.

Online only: Radiance, luminance and brightness — https://blog.shyandram.dev/en/learn/image-processing/fundamentals/notes/radiance-luminance-and-brightness/

Image sensing and acquisition

In plain words: a sensor is a tiny light meter that turns light into voltage. Put many of them in a row or a grid and you have a scanner or a camera.

Single, line and array sensors

Every imaging device combines an **illumination source**, the **scene**, and a **sensor** whose material responds to the energy of interest [1]. The sensor's output voltage is digitized to give one number. Sensors are arranged in three ways:

- **Single sensor.** One detector (for example a photodiode) plus mechanical motion in two directions sweeps out an image one sample at a time. Slow, but cheap and very precise; used in some high-precision scanners.
- **Line (strip) sensor.** A row of detectors captures one line at a time, and motion perpendicular to the strip provides the second dimension. Flatbed scanners and airborne push-broom imagers work this way. A *ring* of sensors around a rotating source is the geometry of computed tomography (CT), where the image is reconstructed rather than read out directly.
- **Array sensor.** A 2-D grid of detectors, such as a CCD or CMOS chip, captures the whole image at once. This is the digital camera. A lens focuses the scene onto the array, each detector integrates light over an exposure time, and the readout electronics produce one value per detector.

Color cameras usually place a mosaic of red, green and blue filters over the array, so each detector measures only one color; the missing two values per pixel are interpolated (demosaicked) in the camera's image-processing pipeline [8].

A simple image formation model

We write a monochrome image as a 2-D function $f(x, y)$ whose value is proportional to the energy arriving at (x, y). Because it comes from physical energy, $0 \le f(x, y) < \infty$. For reflected light, it helps to split f into two factors [1]:

$$f(x, y) = i(x, y)\, r(x, y), \qquad 0 < i(x, y) < \infty, \qquad 0 < r(x, y) < 1,$$

where $i(x, y)$ is the **illumination** (how much light falls on the scene point) and $r(x, y)$ is the **reflectance** (what fraction of it the surface reflects, from total absorption near 0 to total reflection near 1). Illumination depends on the light source; reflectance is a property of the object. For transmission imaging such as X-rays, a **transmissivity** takes the place of reflectance.

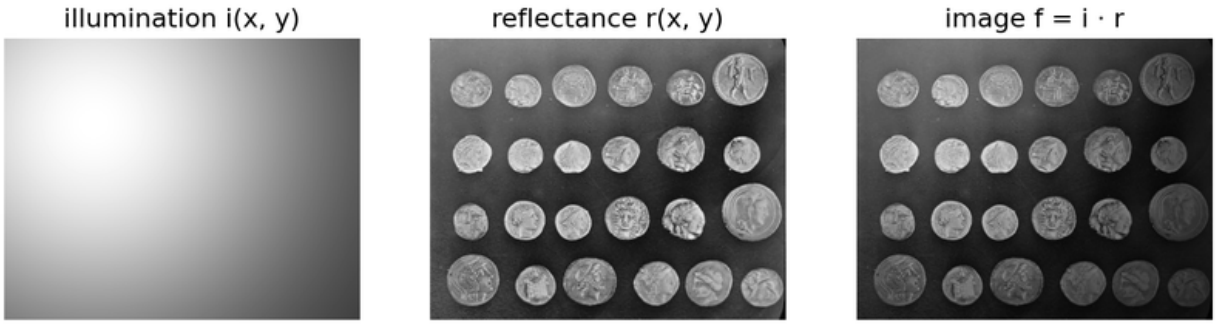


Figure 2.2 — The illumination–reflectance model. A smooth illumination field (left) multiplies the re-

flectance (center) to give the recorded image (right).

```
import numpy as np
from skimage import data, img_as_float

r = img_as_float(data.coins())          # reflectance in [0, 1]
h, w = r.shape
yy, xx = np.mgrid[0:h, 0:w]
i = 0.2 + 0.8 * xx / (w - 1)            # illumination ramp: dim left, bright right
f = i * r                               # what the sensor records
```

The model is simple but useful. Illumination usually varies slowly across a scene, while reflectance changes abruptly at object edges. Later chapters exploit this: homomorphic filtering (Chapter 4) takes $\log f = \log i + \log r$ to separate the two, and shading correction (later in this chapter) divides out a known i.

Image sampling and quantization

In plain words: a camera measures a continuous scene at a finite number of points (sampling) and rounds each measurement to one of a finite number of levels (quantization).

Basic concepts

A sensor's output is a continuous voltage waveform over a continuous plane. To make it digital, both the coordinates and the amplitude must become discrete [1]:

- **Sampling** digitizes the *coordinates*: we keep values only at a grid of positions.
- **Quantization** digitizes the *amplitude*: each kept value is mapped to the nearest of a finite set of levels.

With an array sensor, the number and spacing of detectors fix the sampling; the analog-to-digital converter fixes the quantization. How finely we must sample depends on the image content: by the sampling theorem, a band-limited signal can be reconstructed exactly if it is sampled at more than twice its highest frequency. Chapter 4 develops this properly; Unser's review gives the modern, spline-based view of sampling and reconstruction [2].

Representing digital images

After sampling and quantization, an image is a matrix of M rows and N columns:

$$f = \begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0,N-1) \\ f(1,0) & f(1,1) & \cdots & f(1,N-1) \\ \vdots & \vdots & \ddots & \vdots \\ f(M-1,0) & f(M-1,1) & \cdots & f(M-1,N-1) \end{bmatrix}$$

Each element is a **pixel** (picture element). Following the book's convention [1], x indexes rows and y indexes columns, with the origin at the top-left corner. (Many libraries, including OpenCV's point APIs, use $(x, y) = (\text{column}, \text{row})$. Always check.)

The number of intensity levels is normally a power of two, $L = 2^k$, so values lie in $[0, L-1]$. Storing the image takes

$$b = M \times N \times k \text{ bits},$$

where k is the number of bits per pixel. A 512×512 image with $k = 8$ needs 262,144 bytes. The **dynamic range** of an imaging system is the ratio of the largest measurable intensity (set by saturation) to the smallest (set by noise); the **contrast** of a particular image is the difference between its highest and lowest intensities.

Spatial and intensity resolution

Spatial resolution is the size of the smallest discernible detail. It is meaningful only with units: line pairs per millimeter, or dots per inch (dpi) in printing. Saying an image is "1024 × 1024" says nothing about resolution until we know the physical area it covers. **Intensity resolution** is the smallest discernible change in level, usually stated as the number of bits k.

```
import numpy as np
from skimage import data

img = data.camera()                       # 512 x 512, uint8 (k = 8)
small = img[::4, ::4]                     # keep every 4th sample -> 128 x 128

def quantize(x, bits):
    step = 2 ** (8 - bits)
    return (x // step) * step + step // 2 # map to the centre of each bin

q3 = quantize(img, 3)                     # 8 gray levels
print(small.shape, np.unique(q3).size)    # (128, 128) 8
```

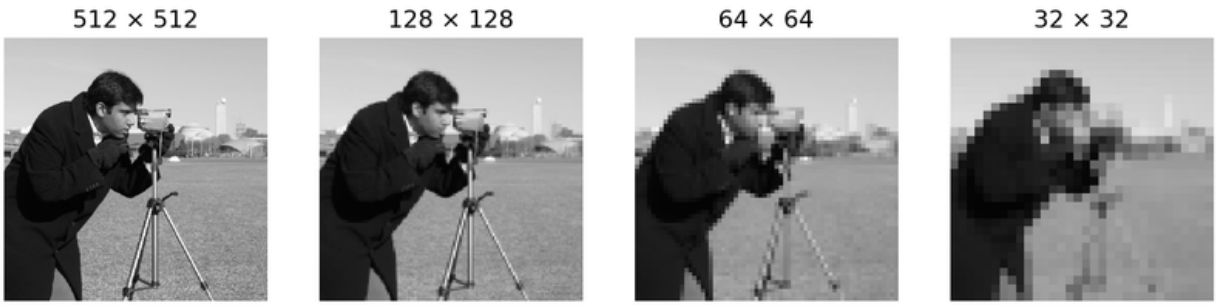


Figure 2.3 — Reducing spatial resolution. Each version is displayed at the same size, so fewer samples show up as larger, blockier pixels.

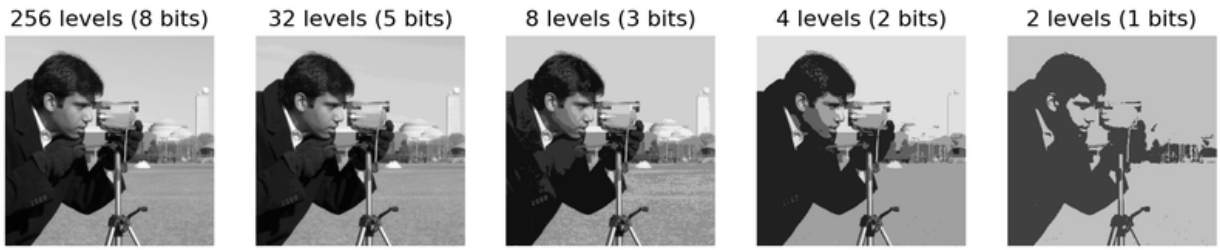


Figure 2.4 — Reducing intensity resolution. Around 8 levels, smooth regions such as the sky develop false contours; at 2 levels only a silhouette remains.

Two lessons follow. Reducing samples first destroys fine texture (grass, hair), then edges. Reducing levels first damages *smooth* regions: the sky in Figure 2.4 breaks into visible steps called **false contouring**, exactly where the Weber ratio says the eye is most sensitive. Images with a lot of detail tolerate fewer gray levels than smooth ones, because busy texture masks the steps [1].

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

Image interpolation

Interpolation estimates values at positions where we have no samples. It is used whenever an image is resized, rotated, warped or registered [1], [3], [4].

Nearest neighbor copies the value of the closest sample. It is fast and keeps the original values, but it makes edges blocky and, when shrinking, can drop thin structures entirely.

Bilinear interpolation uses the four nearest samples. For a point at fractional offsets $a, b \in [0, 1)$ from the top-left sample f_{00},

$$v = (1-a)(1-b)f_{00} + a(1-b)f_{01} + (1-a)bf_{10} + abf_{11},$$

where f_{01}, f_{10} and f_{11} are the right, lower and lower-right neighbors. Equivalently, $v(x, y) = c_1 x + c_2 y + c_3 xy + c_4$, with the four coefficients fixed by the four samples. The result is continuous but slightly blurred.

```
import numpy as np

def bilinear(img, x, y):
    """Sample img (H, W) at real-valued column x, row y."""
    if img.ndim != 2 or not img.size:
        raise ValueError("img must be a nonempty 2-D array")
    if not (0 <= x <= img.shape[1] - 1 and 0 <= y <= img.shape[0] - 1):
        raise ValueError("sample coordinates must lie inside the image")
    x0, y0 = int(np.floor(x)), int(np.floor(y))
    x1, y1 = min(x0 + 1, img.shape[1] - 1), min(y0 + 1, img.shape[0] - 1)
    a, b = x - x0, y - y0
    top = (1 - a) * img[y0, x0] + a * img[y0, x1]
```

```
    bot = (1 - a) * img[y1, x0] + a * img[y1, x1]
    return (1 - b) * top + b * bot

print(bilinear(np.array([[0., 10.], [20., 30.]]), 0.5, 0.5))   # 15.0
```

This helper samples only inside the image, including its last row and column. It rejects out-of-bounds coordinates rather than accidentally wrapping negative NumPy indices. A warp that needs values outside the input must choose a border policy, such as a constant value or reflection.

Bicubic interpolation uses the 16 nearest samples. It weighs them with a smooth piecewise-cubic kernel, most often Keys' cubic convolution kernel [3], applied along the rows and then along the columns. The kernel has small negative lobes, so bicubic keeps edges crisper than bilinear, but it can add a slight overshoot (ringing) next to strong edges.

Online only: The cubic convolution kernel — https://blog.shyandram.dev/en/learn/image-processing/fundamentals/notes/the-cubic-convolution-kernel/

```
import cv2
from skimage import data

crop = data.camera()[150:214, 230:294]    # 64 x 64
up = {name: cv2.resize(crop, None, fx=4, fy=4, interpolation=flag)
      for name, flag in [("nearest", cv2.INTER_NEAREST),
                         ("bilinear", cv2.INTER_LINEAR),
                         ("bicubic", cv2.INTER_CUBIC)]}
```

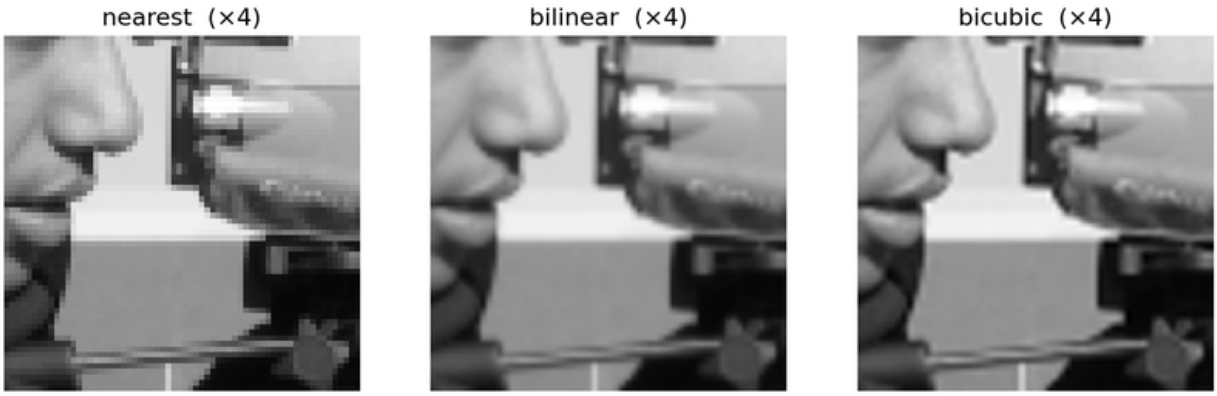


Figure 2.5 — Four-times enlargement of the same crop. Nearest neighbor is blocky; bilinear is smooth but soft; bicubic keeps edges a little crisper.

Drag the slider to compare nearest-neighbor and bicubic upsampling of a 96 × 96 crop enlarged to 384 × 384. Look at the camera's edges and the face profile.



Nearest



Bicubic

Cameraman crop upsampled four times with nearest-neighbor and with bicubic interpolation

No interpolation method can create detail that sampling threw away. Thévenaz, Blu and Unser's review [4] makes the point precisely: the quality of an interpolator is governed by its approximation order and by how closely its kernel approaches the ideal (sinc) reconstructor, and higher-order B-spline methods outperform classic cubic convolution at similar cost. Learned super-resolution, discussed in the Modern view, goes further by *hallucinating* plausible detail from training data.

Basic relationships between pixels

In plain words: before we can talk about objects in an image, we need rules for which pixels "touch" each other and how far apart two pixels are.

Neighbors of a pixel

A pixel p at (x, y) has four horizontal and vertical neighbors,

$$N_4(p) = \{(x+1, y),\ (x-1, y),\ (x, y+1),\ (x, y-1)\},$$

four diagonal neighbors,

$$N_D(p) = \{(x+1, y+1),\ (x+1, y-1),\ (x-1, y+1),\ (x-1, y-1)\},$$

and the eight neighbors $N_8(p) = N_4(p) \cup N_D(p)$ [1]. At the image border some neighbors fall outside the image; how we treat them (ignore, pad, reflect) is a choice we meet again in filtering.

Adjacency, connectivity, regions and boundaries

Let V be the set of intensity values that count as "similar" — for a binary image, $V = \{1\}$; for a gray image, perhaps all values between 100 and 120. Two pixels p and q with values in V are [1]:

- **4-adjacent** if $q \in N_4(p)$;
- **8-adjacent** if $q \in N_8(p)$;
- **m-adjacent** (mixed) if $q \in N_4(p)$, *or* $q \in N_D(p)$ and the set $N_4(p) \cap N_4(q)$ contains no pixel whose value is in V.

Why m-adjacency? With 8-adjacency, a diagonal step and a two-step "around the corner" route can both connect the same pair of pixels, so paths become ambiguous. m-adjacency keeps a diagonal link only when no 4-connected route exists, which removes the redundant loops (Figure 2.6, right).



Figure 2.6 — Neighborhoods of a pixel p, and m-adjacency on a binary patch: the dashed diagonal is not an m-adjacency because the two pixels already share a 4-neighbor in V.

A **digital path** from p to q is a sequence of distinct pixels $(x_0, y_0), \dots, (x_n, y_n)$ with consecutive pixels adjacent; n is the length, and the path is **closed** if the first and last pixels coincide. Depending on the adjacency used we speak of 4-, 8- or m-paths. For a subset S of pixels, p and q are **connected in** S if a path between them lies entirely in S. The set of pixels connected to p in S is a **connected component**; if S has only one, it is a **connected set**.

A connected set is called a **region**. Two regions are adjacent if their union is connected. The **boundary** (or border) of a region R is the set of pixels in R that have at least one neighbor outside R. Note the difference between a boundary and an **edge**: a boundary is a global, closed property of a region, while an edge is a local intensity discontinuity that may not close at all (Chapter 10).

The choice of adjacency changes the answer to simple questions such as "how many objects are there?":

```
import numpy as np
from skimage.measure import label

b = np.array([[1, 0, 0],
              [0, 1, 0],
              [0, 0, 1]])
print(label(b, connectivity=1).max(),     # 4-connectivity: 3 objects
      label(b, connectivity=2).max())     # 8-connectivity: 1 object
```

A classic consequence: if the foreground uses 8-connectivity, the background should use 4-connectivity (and vice versa), otherwise a closed digital curve may fail to separate its inside from its outside. This is a basic result of digital topology [1].

Distance measures

For pixels $p = (x, y)$, $q = (u, v)$ and z, a function D is a **distance metric** if $D(p, q) \geq 0$ with equality only when $p = q$, $D(p, q) = D(q, p)$, and $D(p, z) \leq D(p, q) + D(q, z)$. Three metrics are standard [1], [5]:

$$
\begin{aligned}
D_E(p,q) &= \sqrt{(x-u)^2 + (y-v)^2} && \text{(Euclidean)} \\
D_4(p,q) &= |x-u| + |y-v| && \text{(city-block)} \\
D_8(p,q) &= \max(|x-u|,\ |y-v|) && \text{(chessboard)}
\end{aligned}
$$

The pixels within a fixed D_E of p form a disk; within a fixed D_4, a diamond; within a fixed D_8, a square. The pixels with $D_4 = 1$ are exactly $N_4(p)$, and those with $D_8 = 1$ are exactly $N_8(p)$. D_4 and D_8 depend only on coordinates, not on pixel values; a distance *along* m-paths, by contrast, does depend on the values because the paths do.

```
import numpy as np
p, q = np.array([2, 3]), np.array([5, 7])
d = np.abs(p - q)
print("D_E =", np.hypot(*d), " D_4 =", d.sum(), " D_8 =", d.max())   # 5.0, 7, 4
```

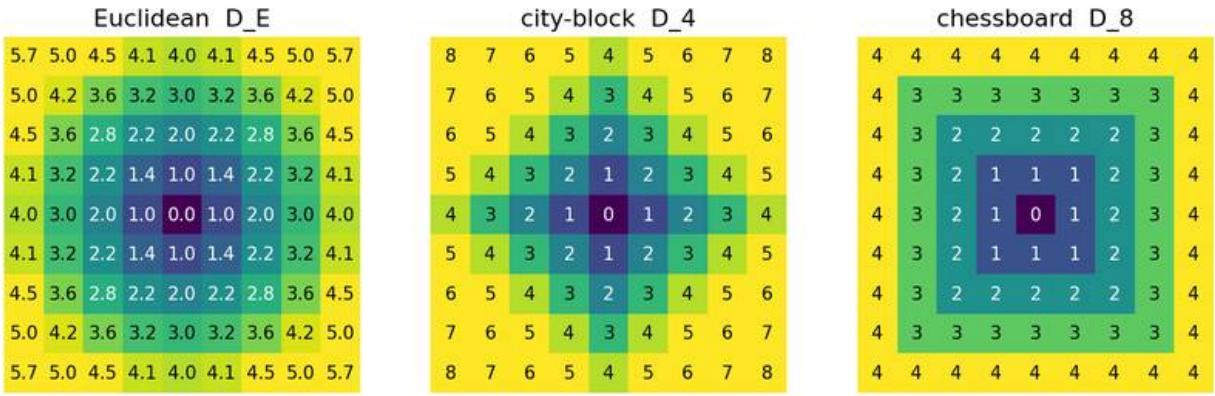


Figure 2.7 — Distance from the center pixel under the three metrics. Equal-distance contours are circles, diamonds and squares.

Rosenfeld and Pfaltz [5] introduced efficient two-pass algorithms to compute such distances to the nearest object pixel for every pixel at once — the **distance transform**, still used for skeletons, morphology and shape matching (Chapter 9).

Introduction to the mathematical tools used in DIP

In plain words: an image is a matrix, so most image processing is arithmetic on matrices. This section lists the operations we will use again and again, and the one property — linearity — that decides which tools apply.

Elementwise versus matrix operations

An **elementwise** (array) operation acts pixel by pixel. For two 2×2 images the elementwise product is

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \odot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{22}b_{22} \end{bmatrix},$$

whereas the matrix product follows row-times-column rules. Unless stated otherwise, image operations — including "multiplying two images" — are elementwise [1]. In NumPy, `A * B` is elementwise and `A @ B` is the matrix product.

Linear versus nonlinear operations

Consider an operator $\mathcal{H}$ that maps an input image f to an output $g = \mathcal{H}[f]$. It is **linear** if

$$\mathcal{H}[af_1 + bf_2] = a\,\mathcal{H}[f_1] + b\,\mathcal{H}[f_2]$$

for all images f_1, f_2 and all constants a, b. The two parts of this property are **additivity** (the output of a sum is the sum of outputs) and **homogeneity** (scaling the input scales the output). Summing all pixels is linear; taking the maximum is not:

```
import numpy as np
rng = np.random.default_rng(0)
f1, f2 = rng.random((4, 4)), rng.random((4, 4))
a, b = 2.0, -3.0
print(np.isclose(np.sum(a*f1 + b*f2), a*np.sum(f1) + b*np.sum(f2)))   # True
print(np.isclose(np.max(a*f1 + b*f2), a*np.max(f1) + b*np.max(f2)))   # False
```

Linearity matters because linear operators are fully described by their response to simple inputs, and a large body of theory (convolution, the Fourier transform, Chapters 3–5) applies to them. Nonlinear operators such as the median filter or morphology can do things linear ones cannot, but are harder to analyze.

Arithmetic operations

Sum, difference, product and quotient of two images of the same size are all elementwise [1]. Three uses stand out.

Averaging noisy images. Suppose we capture K images of a static scene, $g_i(x,y) = f(x,y) + \eta_i(x,y)$, where f is the clean image and η_i is noise that is uncorrelated between frames, has zero mean and variance σ_η^2. The average $\bar{g} = \frac{1}{K}\sum_{i=1}^{K} g_i$ satisfies

$$E\{\bar{g}(x,y)\} = f(x,y), \qquad \sigma_{\bar{g}}^2(x,y) = \frac{1}{K}\,\sigma_\eta^2(x,y),$$

so the noise standard deviation falls as $1/\sqrt{K}$. Astronomers stack exposures for this reason, and smartphone "night modes" do the same after aligning a burst of frames [7].

```
import numpy as np
```

```
from skimage import data, img_as_float

g = img_as_float(data.camera())
rng = np.random.default_rng(1)
for K in (1, 4, 16, 64):
    avg = np.mean([g + rng.normal(0, 0.2, g.shape) for _ in range(K)], axis=0)
    print(K, round(float(np.std(avg - g)), 3))    # 0.2, 0.1, 0.05, 0.025
```

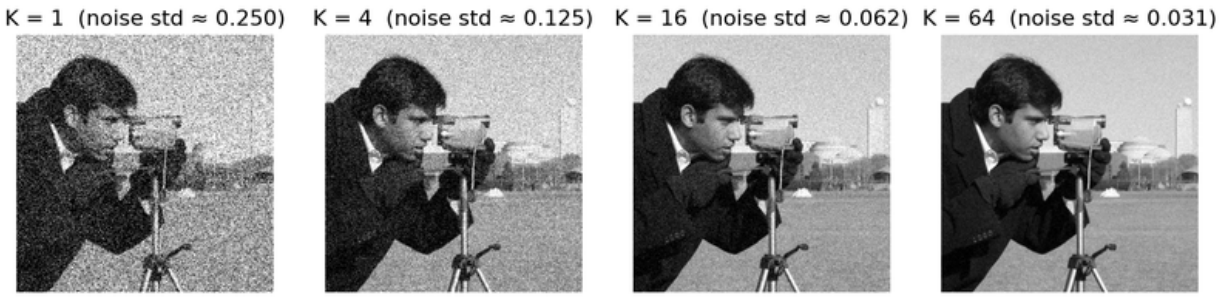


Figure 2.8 — Averaging K independent noisy frames. The noise standard deviation drops by a factor of about two each time K is multiplied by four.

Subtraction reveals differences. Subtracting two frames of a video highlights motion; in medical imaging, subtracting a pre-contrast "mask" image from a post-contrast image leaves mainly the vessels that took up the contrast agent.

Multiplication and division handle shading and masking. If a sensor's shading pattern $h(x, y)$ is known (for example by imaging a uniform target), the observed image $g = f \cdot h$ can be corrected as $f = g/h$. Multiplying by a binary mask keeps a region of interest and zeroes the rest.

One practical rule: arithmetic results often leave the valid range (differences can be negative, sums can exceed 255). Do the arithmetic in floating point, then rescale. A simple rescaling to $[0, K]$ is $f_m = f - \min f$ followed by $f_s = K f_m / \max f_m$.

Set and logical operations

Treat a binary image as a **set** A of pixel coordinates (the foreground). The usual set operations then apply [1]: union $A \cup B$, intersection $A \cap B$, complement $A^c = \{w \mid w \notin A\}$, and difference $A - B = A \cap B^c$. On binary arrays these become the logical operations OR, AND, NOT and AND-NOT. For gray images, union and intersection are commonly defined as elementwise maximum and minimum. Fuzzy sets generalize this further by letting membership vary between 0 and 1.

```
import numpy as np
from skimage import data

coins = data.coins()
A = coins > 100                                        # set A: bright pixels
B = np.zeros_like(A); B[:, : coins.shape[1] // 2] = True   # set B: left half
union, inter, diff = A | B, A & B, A & ~B
```

Set operations are the language of mathematical morphology (Chapter 9) and of combining segmentation masks.

Spatial operations

Spatial operations act directly on pixel values and positions. There are three kinds [1].

Single-pixel operations change each value independently: $s = T(z)$, where z is the input intensity and s the output. The image negative $s = (L-1) - z$ is an example. These are the intensity transformations of Chapter 3.

Neighborhood operations compute each output from a neighborhood S_{xy} around (x, y). Local averaging,

$$g(x,y) = \frac{1}{mn} \sum_{(r,c) \in S_{xy}} f(r,c),$$

where S_{xy} is an $m \times n$ window centered at (x, y), is the simplest; Chapter 3 generalizes it to spatial filtering.

Geometric transformations move pixels. They have two steps: a spatial transformation of coordinates, and intensity interpolation to fill the new grid. The most common family is the **affine** transform. In homogeneous coordinates,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix},$$

where (x, y) is an input location, (x', y') the output location, the a_{ij} encode rotation, scaling and shear, and (t_x, t_y) is a translation. Affine maps preserve straight lines and parallelism, and chaining several is just multiplying their matrices.

Implementations almost always use **inverse mapping**: for each output pixel, apply the inverse transform to find where it came from in the input, and interpolate there. Forward mapping (pushing each input pixel to the output) leaves holes and collisions.

```
import cv2
from skimage import data

img = data.camera()
h, w = img.shape
M = cv2.getRotationMatrix2D(center=(w / 2, h / 2), angle=30, scale=0.8)  # 2x3 affine
rot = cv2.warpAffine(img, M, (w, h), flags=cv2.INTER_LINEAR)  # inverse mapping inside
```

Two related tasks build on these transforms. **Registration** aligns two images of the same scene when the transform is unknown, by estimating it from points that can be found in both images. A **homography** is a 3×3 transform that adds perspective to the affine one; it relates two photos of the same flat surface taken from different viewpoints.

Online only: Image registration — *https://blog.shyandram.dev/en/learn/image-processing/fundamentals/notes/image-registration/*

Online only: Homography: the projective transform — *https://blog.shyandram.dev/en/learn/image-processing/fundamentals/notes/homography-the-projective-transform/*

Vector and matrix operations

A color pixel is a vector, $\mathbf{z} = [z_1, z_2, z_3]^T$ for its red, green and blue values. Vector tools then apply directly; for example, the Euclidean distance from $\mathbf{z}$ to a reference color $\mathbf{a}$,

$$D(\mathbf{z}, \mathbf{a}) = \left[(\mathbf{z} - \mathbf{a})^T (\mathbf{z} - \mathbf{a})\right]^{1/2},$$

is the basis of simple color segmentation (Chapter 6). Stacking an $M \times N$ image into an $MN \times 1$ vector also lets any linear operation be written as $\mathbf{g} = \mathbf{Hf}$, a form used in restoration (Chapter 5).

Image transforms

Some problems are easier in a different domain. A 2-D linear transform takes the general form [1]

$$T(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y)\, r(x, y, u, v),$$

where f is the input image, r is the **forward transformation kernel**, and (u, v) are the transform-domain variables. The inverse uses an inverse kernel $s(x, y, u, v)$. When $r(x, y, u, v) = r_1(x, u)\, r_2(y, v)$ the kernel is **separable**, and the 2-D transform can be done as 1-D transforms along rows and then columns, or in matrix form $\mathbf{T} = \mathbf{AFB}$. The Fourier transform (Chapter 4) and the wavelet transforms (Chapter 7) are the two you will meet most.

Probability methods

Intensities can be treated as random quantities. If n_k pixels in an $M \times N$ image have intensity z_k, the probability of that intensity is estimated by

$$p(z_k) = \frac{n_k}{MN}, \qquad \sum_{k=0}^{L-1} p(z_k) = 1,$$

which is the **normalized histogram**. The mean and variance of intensities are

$$m = \sum_{k=0}^{L-1} z_k\, p(z_k), \qquad \sigma^2 = \sum_{k=0}^{L-1} (z_k - m)^2\, p(z_k),$$

where m measures average brightness and σ (the standard deviation) measures contrast. Higher moments describe skew and peakedness.

```
import numpy as np
from skimage import data

img = data.camera()
p = np.bincount(img.ravel(), minlength=256) / img.size   # p(z_k)
z = np.arange(256)
m = np.sum(z * p)
sigma = np.sqrt(np.sum((z - m) ** 2 * p))
print(round(m, 2), round(sigma, 2))                      # 129.06 73.64
```

Histogram processing (Chapter 3) and noise modeling (Chapter 5) both start here.

Modern view

The fundamentals in this chapter have not been replaced by deep learning; they have become the *interface* to it. Networks consume sampled, quantized arrays, are trained with losses that encode a model of perception, and are evaluated on data produced by camera pipelines. The reviews below map this territory.

Sampling and interpolation theory. Unser's review [2] treats sampling as approximation with a practical basis, such as B-splines, instead of demanding ideal band-limited signals and sinc reconstruction. Its takeaway: a good prefilter before sampling and a good reconstruction kernel after it matter more than the exact sampling rate. Thévenaz, Blu and Unser [4] compare interpolation kernels on equal terms and find that B-spline interpolation gives a better quality-for-cost trade-off than Keys' cubic convolution [3].

Learned super-resolution. Wang, Chen and Hoi [9] and Moser et al. [10] survey deep super-resolution, from CNN designs to transformers and diffusion models. Two methods show classical ideas surviving inside networks: LIIF [11] is a learned interpolator that can be queried at any coordinate, and Real-ESRGAN [12] synthesizes training data by applying blur, noise, resizing and compression more than once, because training pairs made by bicubic downsampling do not match real degradations. The lesson: bicubic interpolation is both the baseline and a hidden assumption in training data.

Perception and image quality. SSIM [13] turned the observations of the perception section into a quality index that compares local luminance, contrast and structure; Zhai and Min [14] survey the whole field. Distances between deep network features (LPIPS) agree with human similarity judgments far better than PSNR or SSIM [15], and Ding et al. [16] found that a metric's ranking on quality benchmarks does not reliably predict how useful it is as a training loss. In short, Mach bands and the Weber ratio now live inside loss functions.

Online only: MSE, PSNR and the data range — https://blog.shyandram.dev/en/learn/image-processing/fundamentals/notes/mse-psnr-and-the-data-range/

Camera sensor pipelines. Ramanath et al. [8] describe the classic in-camera pipeline (demosaicking, white balance, color transforms, gamma, compression). Delbracio et al. [7] show how phones

reworked it with burst capture, alignment and merging, which is the averaging-noisy-images idea at scale. Learned models now replace parts of the pipeline or all of it [17], [18].

Online only: The reviews in more detail — https://blog.shyandram.dev/en/learn/image-processing/fundamentals/papers/the-reviews-in-more-detail/

What did *not* change? The pixel grid, the bit depth, adjacency and connectivity (still the basis of connected-component labeling in every segmentation post-processing step), affine warps (now differentiable layers inside networks), and histograms. Knowing them is what lets you spot when a model's input has been resized with the wrong kernel, quantized too coarsely, or compared with a metric that does not match human judgment.

Key takeaways

- Human vision senses *relative* changes (Weber's law) and exaggerates edges (Mach bands); perceived brightness depends on the surround. Good image processing respects these facts.
- Images can be formed from any part of the electromagnetic spectrum; for reflected light, $f = i \cdot r$ separates illumination from reflectance.
- Sampling sets spatial resolution, quantization sets intensity resolution, and storage is MNk bits. Too few levels cause false contours in smooth regions.
- Interpolation — nearest, bilinear, bicubic — estimates values between samples; it trades speed for smoothness and sharpness but cannot recover lost detail.
- 4-, 8- and m-adjacency define paths, connected components, regions and boundaries; D_4 and D_8 are distance metrics whose unit balls are diamonds and squares.
- Image operations are elementwise unless stated otherwise; linearity decides whether convolution and Fourier tools apply.
- Averaging K independent noisy frames reduces noise standard deviation by $\sqrt{K}$; geometric transforms use inverse mapping plus interpolation.
- Deep learning built on these fundamentals: learned interpolators, perceptual losses and learned camera pipelines all inherit this chapter's concepts.

Exercises

1. A grayscale scanner produces 2400 × 3300 images at 12 bits per pixel. How many megabytes does one uncompressed scan need? How many if it is reduced to 8 bits?

   Hint

   Use $b = MNk$ and divide by 8×10^6 (or 8×2^{20} for MiB). At 12 bits: $2400 \cdot 3300 \cdot 12/8 = 11.88$ MB. At 8 bits: 7.92 MB.

2. Quantize `skimage.data.moon()` to 4, 5 and 6 bits. At which level do false contours disappear for you? Then add a small amount of uniform random noise (amplitude half a quantization step) *before* quantizing at 4 bits. What changes, and why?

Hint

The noise (called dither) breaks up the large flat steps, so the eye averages the fine noise instead of seeing contours. The mean error stays similar but the visible structure changes — another reminder that squared error and perceived quality differ.

3. In the binary image below, count the connected components of 1-pixels under 4-, 8- and m-connectivity. Then count the components of 0-pixels under 4- and 8-connectivity. Which pairing gives a consistent "inside/outside" answer?

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

Hint

The 1s are a single component under all three adjacencies, but the ring is *closed* only if diagonal steps count, because of the gap at the lower right. The center 0 is isolated under 4-connectivity (a hole) but joins the outside under 8-connectivity. 4/4 gives an open curve with a hole; 8/8 gives a closed curve without one. Only 8-foreground with 4-background (or the reverse) is consistent. Check your answer with `skimage.measure.label`.

4. Show that $D_8(p,q) \le D_E(p,q) \le D_4(p,q)$ for any two pixels, and find pixel pairs where each inequality becomes an equality.

Hint

Let $d_x = |x-u|$, $d_y = |y-v|$. Then $\max(d_x,d_y)^2 \le d_x^2 + d_y^2 \le (d_x+d_y)^2$. Equality holds when one of d_x, d_y is zero.

5. Write a function that rotates an image by θ using *forward* mapping (push each input pixel to its rounded output position). Display the result for $\theta = 17°$ and explain the holes. Then fix it with inverse mapping and bilinear interpolation.

Hint

A rotation preserves distances, but rotated pixel centers generally fall between output-grid points. Rounding a forward mapping can leave holes or send multiple samples to one pixel. Inverse mapping visits every output pixel and interpolates at its source coordinate; if that coordinate falls outside the image, an explicit border rule is still needed.

6. You have 10 noisy frames of a static scene with noise standard deviation 12 gray levels. How many frames would you need to bring it below 2 gray levels? What assumption could fail in a real handheld burst, and how do mobile pipelines handle it?

Hint

$12/\sqrt{K} < 2$ requires $K > 36$, so 37 frames. The assumption that the scene and camera are static fails; frames must be aligned (registered) before averaging, and moving objects need robust merging [7].

References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018, Ch. 2. publisher page
2. M. Unser, "Sampling—50 years after Shannon," *Proceedings of the IEEE*, vol. 88, no. 4, pp. 569–587, 2000. doi
3. R. Keys, "Cubic convolution interpolation for digital image processing," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 6, pp. 1153–1160, 1981. doi
4. P. Thévenaz, T. Blu, and M. Unser, "Interpolation revisited," *IEEE Transactions on Medical Imaging*, vol. 19, no. 7, pp. 739–758, 2000. doi
5. A. Rosenfeld and J. L. Pfaltz, "Distance functions on digital pictures," *Pattern Recognition*, vol. 1, pp. 33–61, 1968. doi
6. B. Zitová and J. Flusser, "Image registration methods: a survey," *Image and Vision Computing*, vol. 21, no. 11, pp. 977–1000, 2003. doi
7. M. Delbracio, D. Kelly, M. S. Brown, and P. Milanfar, "Mobile computational photography: A tour," arXiv:2102.09000, 2021. arXiv
8. R. Ramanath, W. E. Snyder, Y. Yoo, and M. S. Drew, "Color image processing pipeline," *IEEE Signal Processing Magazine*, vol. 22, no. 1, pp. 34–43, 2005. doi
9. Z. Wang, J. Chen, and S. C. H. Hoi, "Deep learning for image super-resolution: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020. arXiv
10. B. Moser, F. Raue, S. Frolov, J. Hees, S. Palacio, and A. Dengel, "Hitchhiker's guide to super-resolution: Introduction and recent advances," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023. arXiv
11. Y. Chen, S. Liu, and X. Wang, "Learning continuous image representation with local implicit image function," CVPR, 2021. arXiv
12. X. Wang, L. Xie, C. Dong, and Y. Shan, "Real-ESRGAN: Training real-world blind super-resolution with pure synthetic data," arXiv:2107.10833, 2021. arXiv
13. Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. project page
14. G. Zhai and X. Min, "Perceptual image quality assessment: A survey," *Science China Information Sciences*, vol. 63, no. 11, 2020. doi
15. R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang, "The unreasonable effectiveness of deep features as a perceptual metric," CVPR, 2018. arXiv
16. K. Ding, K. Ma, S. Wang, and E. P. Simoncelli, "Comparison of full-reference image quality models for optimization of image processing systems," *International Journal of Computer Vision*, 2021. arXiv
17. C. Chen, Q. Chen, J. Xu, and V. Koltun, "Learning to see in the dark," CVPR, 2018. arXiv

18. A. Ignatov, L. Van Gool, and R. Timofte, “Replacing mobile camera ISP with a single deep learning model,” arXiv:2002.05509, 2020. arXiv
19. Q. Huynh-Thu and M. Ghanbari, “Scope of validity of PSNR in image/video quality assessment,” *Electronics Letters*, vol. 44, no. 13, pp. 800–801, 2008. doi
20. R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed., Cambridge University Press, 2004. doi
21. scikit-image contributors, “ProjectiveTransform.estimate,” scikit-image 0.25.2 documentation, accessed 5 November 2026. API documentation.

IP 03 • Intensity Transformations and Spatial Filtering

Remap pixel values with curves and histograms, then smooth and sharpen images with small kernels: the everyday toolbox of spatial-domain enhancement.

Online edition: https://blog.shyandram.dev/en/learn/image-processing/intensity-spatial-filtering/

What you'll learn

- How a single curve $s = T(r)$ can brighten, darken, invert or stretch an image, and when to pick a log, gamma or piecewise-linear curve.
- Why histogram equalization works, derived step by step, and how histogram matching, CLAHE-style local methods and local statistics build on it.
- What linear spatial filtering is, how correlation differs from convolution, and why separable kernels are fast.
- How smoothing (lowpass) filters, including the nonlinear median filter, remove noise, and how derivative-based (highpass) filters sharpen.
- How to derive highpass, bandpass and bandreject kernels from a lowpass one, and how to chain several methods into one pipeline.

The big picture

This chapter is about improving an image without leaving the pixel grid. Every method here works on the pixel values directly, which is what "spatial domain" means. There are two families. **Point operations** change each pixel using only its own value. **Neighborhood operations** (spatial filters) change each pixel using the values around it.

> **Explain like I'm five**
>
> Imagine a box of crayon drawings that are all too pale. A **point operation** is like a magic rule for every crayon color: "light pink becomes red, light blue becomes dark blue." You look at one spot, check the rule, and recolor it. A **filter** is like looking at a spot *and its neighbors* before you decide. If you want the drawing softer, you color each spot as the average of its neighbors. If you want the outlines crisper, you make each spot *more different* from its neighbors.

Why it matters: these are the first tools anyone reaches for, in phone cameras, medical viewers, microscopes and as preprocessing for computer-vision models. They are also the conceptual ancestors of the convolutional layer. A CNN's first layer is a bank of spatial filters, and many learned enhancement networks still output a curve or a filter, as we will see in the Modern view.

Background

Plain version. We write the output image as some operation applied to the input image. If the operation only looks at one pixel at a time, it is a lookup table. If it looks at a small window, it is a filter.

Precise version. Let $I(x, y)$ be the input image and $G(x, y)$ the output, with x, y integer pixel coordinates (the same names as in the convolution tutorial). A spatial-domain process is

$$G(x, y) = T[I](x, y),$$

where T is an operator defined over a **neighborhood** of (x, y). The neighborhood is usually a small square or rectangle centered on (x, y). The operator is applied at every location by moving the neighborhood's center from pixel to pixel.

When the neighborhood is a single pixel (1×1), T depends only on the value at that pixel. We then call it an **intensity transformation** (also gray-level or point transformation) and write it with scalars:

$$s = T(r),$$

where r is the input intensity at a pixel, s is the output intensity at the same pixel, and both lie in $[0, L - 1]$ for an image with L intensity levels ($L = 256$ for 8-bit images). Because r takes only L values, any T can be stored as a lookup table of length L, which is why point operations are extremely fast.

Basic intensity transformation functions

Plain version. Draw a curve from input brightness (horizontal axis) to output brightness (vertical axis). Where the curve is steep, nearby gray levels get pulled apart, so contrast increases. Where it is flat, they get squeezed together, so contrast drops.

The slope $T'(r)$ is the local contrast gain: a small difference Δr between two neighboring pixels becomes roughly $T'(r)\,\Delta r$ after the transformation. Every curve below is a choice about where to spend that gain.

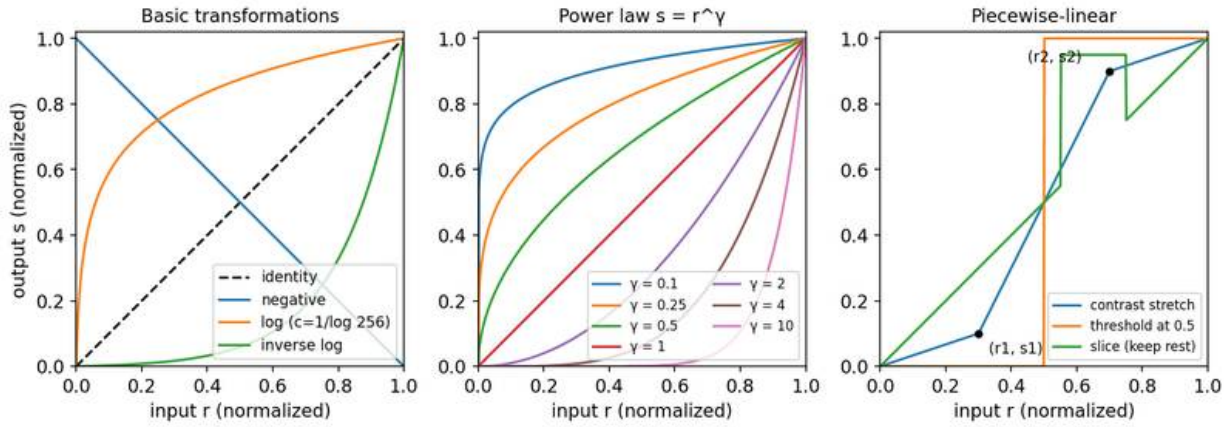


Figure 3.1 — Common intensity transformation curves, with intensities normalized to $[0, 1]$. Left: negative, log and inverse log. Middle: power-law (gamma) curves. Right: piecewise-linear curves.

Image negatives

$$s = (L - 1) - r.$$

Dark becomes light and light becomes dark. The slope is -1 everywhere, so contrast magnitude is unchanged; only the polarity flips. Negatives help when small bright details sit inside large dark regions, because people often judge gray detail more easily on a light background.

Log transformations

$$s = c \log(1 + r),$$

where $c > 0$ is a scale constant, usually chosen so that the largest input maps to $L - 1$, that is $c = (L - 1)/\log L$. The $+1$ keeps $\log 0$ out of the formula. The slope $c/(1 + r)$ is large for small r and small for large r. So the log curve expands dark values and compresses bright ones.

Its classic use is displaying data with a huge dynamic range, such as the magnitude of a Fourier spectrum, whose values can span many orders of magnitude. Scaled linearly, only the brightest point is visible. After the log, the structure appears (Figure 3.2, bottom row). The **inverse log** (exponential) curve does the opposite.

Power-law (gamma) transformations

$$s = c\, r^{\gamma},$$

where $c > 0$ and $\gamma > 0$ are constants (with r normalized to $[0, 1]$, take $c = 1$). For $\gamma < 1$ the curve bows upward and brightens the dark range, like the log. For $\gamma > 1$ it bows downward and darkens it. Unlike the log, one parameter gives a whole family of curves (Figure 3.1, middle).

Gamma matters beyond enhancement. Displays and cameras have power-law responses, and image files usually store gamma-encoded values rather than values proportional to light. Pre-compensating for a device's response is called **gamma correction**. One practical consequence: averaging, blurring or resizing gamma-encoded pixels is not the same as doing it on linear light, which is why careful pipelines linearize first.

```
import numpy as np
from skimage import data, img_as_float

r = img_as_float(data.camera())        # intensities scaled to [0, 1]
negative = 1.0 - r                     # s = (L-1) - r, in normalized form
log_t = np.log1p(r) / np.log(2.0)      # s = c log(1 + r), c chosen so s(1) = 1
bright = r ** 0.4                      # gamma < 1 lifts the shadows
dark = r ** 2.5                        # gamma > 1 deepens them
```

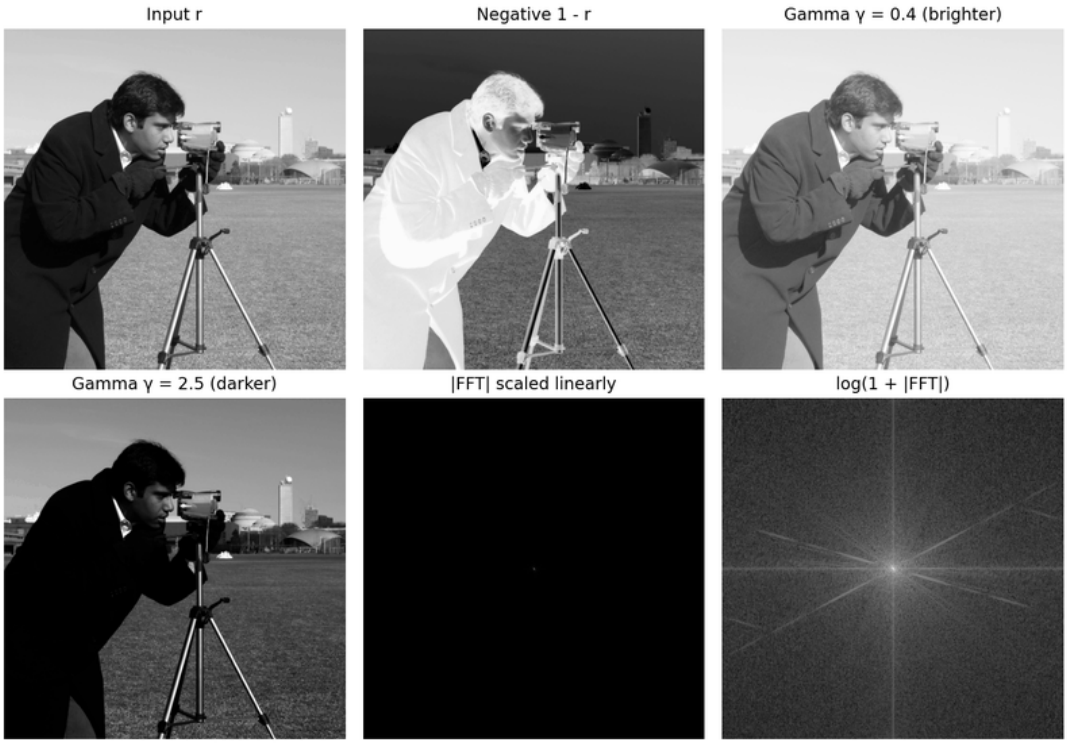


Figure 3.2 — Top: input, negative, and gamma 0.4. Bottom: gamma 2.5; a Fourier magnitude scaled linearly (almost all black); and the same magnitude after $\log(1 + \cdot)$.

Piecewise-linear transformations

Instead of one formula, we can join straight segments. This gives precise control, at the cost of more parameters.

Contrast stretching. Choose two control points (r_1, s_1) and (r_2, s_2) and connect $(0,0) \to (r_1, s_1) \to (r_2, s_2) \to (L-1, L-1)$ with straight lines. The middle segment has slope $(s_2 - s_1)/(r_2 - r_1)$. If $r_1 \le r_2$ and $s_1 \le s_2$, the curve is monotonic and never reverses the order of gray levels. Two special cases are worth remembering:

- $r_1 = r_{\min}$, $s_1 = 0$, $r_2 = r_{\max}$, $s_2 = L - 1$ maps the image's actual range onto the full range (a **min–max stretch**). In practice, use the 1st and 99th percentiles instead of the min and max, so a few outlier pixels do not control the result.
- $r_1 = r_2 = m$, $s_1 = 0$, $s_2 = L - 1$ produces a binary image: a **threshold** at m.

```
def stretch(img, r1, s1, r2, s2, L=256):
    lut = np.interp(np.arange(L), [0, r1, r2, L - 1], [0, s1, s2, L - 1])
    return lut.astype(np.uint8)[img]

img = data.camera()
lo, hi = np.percentile(img, [1, 99])
out = stretch(img, lo, 0, hi, 255)     # robust min-max stretch
```

Intensity-level slicing. Highlight one band of intensities $[A, B]$. Either set the band to a high value and everything else to a low value (a binary result), or brighten the band and leave everything else unchanged. Slicing is useful when the gray range of interest is known, such as one tissue type or one material in an X-ray.

Bit-plane slicing. An 8-bit pixel value is a sum of eight bits, and collecting one bit from every pixel gives a binary image called a **bit plane**. The top planes carry the visible structure, while the lowest planes look like noise. Figure 3.3 shows that the top four planes alone rebuild an image very close to the original, which is why bit planes matter for compression and why the lowest planes are a classic place to hide watermarks.

Online only: Bit planes in detail — https://blog.shyandram.dev/en/learn/image-processing/intensity-spatial-filtering/notes/bit-planes-in-detail/

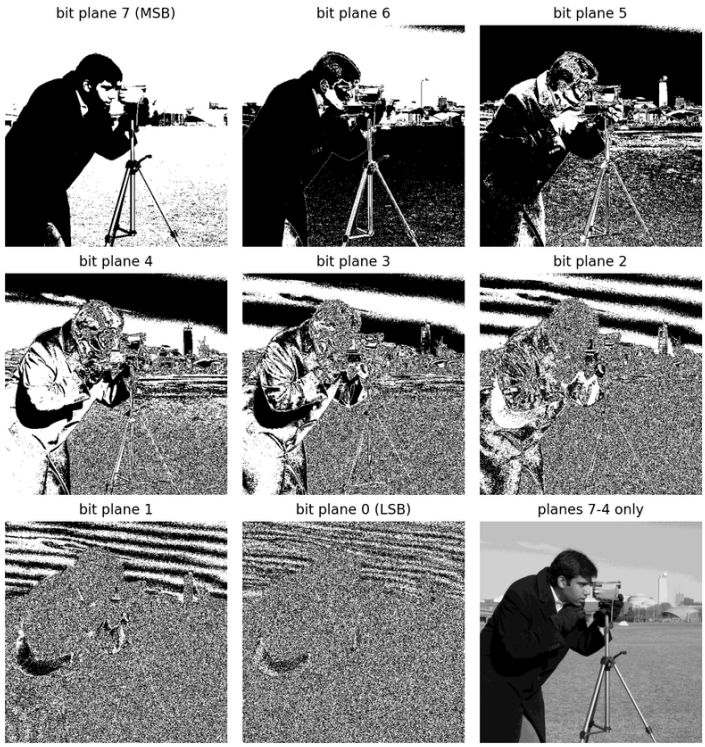


Figure 3.3 — The eight bit planes of an 8-bit image, from the most significant (top left) to the least significant, and the image rebuilt from planes 7–4 only.

Histogram processing

Plain version. A histogram counts how many pixels have each brightness. A dark image has its counts piled up on the left; a washed-out one has them squeezed into a narrow band. Histogram processing chooses a curve T *automatically from those counts*, so that the output histogram has a shape we want.

Precise version. For an $M \times N$ image with levels r_k, $k = 0, \dots, L-1$, let n_k be the number of pixels with value r_k. The **normalized histogram**

$$p_r(r_k) = \frac{n_k}{MN}$$

is an estimate of the probability that a randomly chosen pixel has value r_k. It sums to 1.

Histogram equalization

The goal is a curve T that makes the output histogram as flat as possible, so every gray level is used about equally often.

Derivation (continuous case). Treat intensity as a continuous random variable $r \in [0, L-1]$ with probability density function (PDF) $p_r(r)$. Assume T is strictly increasing on that interval. Then a pixel lands in a small interval $[r, r+dr]$ exactly when its output lands in $[s, s+ds]$ with $s = T(r)$. Equal probabilities give $p_s(s)\,|ds| = p_r(r)\,|dr|$, so

$$p_s(s) = p_r(r)\left|\frac{dr}{ds}\right|.$$

Now choose T as the scaled cumulative distribution function (CDF) of r:

$$s = T(r) = (L-1)\int_0^r p_r(w)\,dw,$$

where w is a dummy integration variable. By the fundamental theorem of calculus, $ds/dr = (L-1)\,p_r(r)$. Substituting,

$$p_s(s) = p_r(r)\cdot\frac{1}{(L-1)\,p_r(r)} = \frac{1}{L-1}, \qquad 0 \le s \le L-1.$$

The output PDF is uniform, whatever the input PDF was. The CDF is the one monotonic curve that spreads probability evenly: it is steep where many pixels live (so they get pulled apart) and flat where few pixels live.

Discrete version. Replace the integral by a sum:

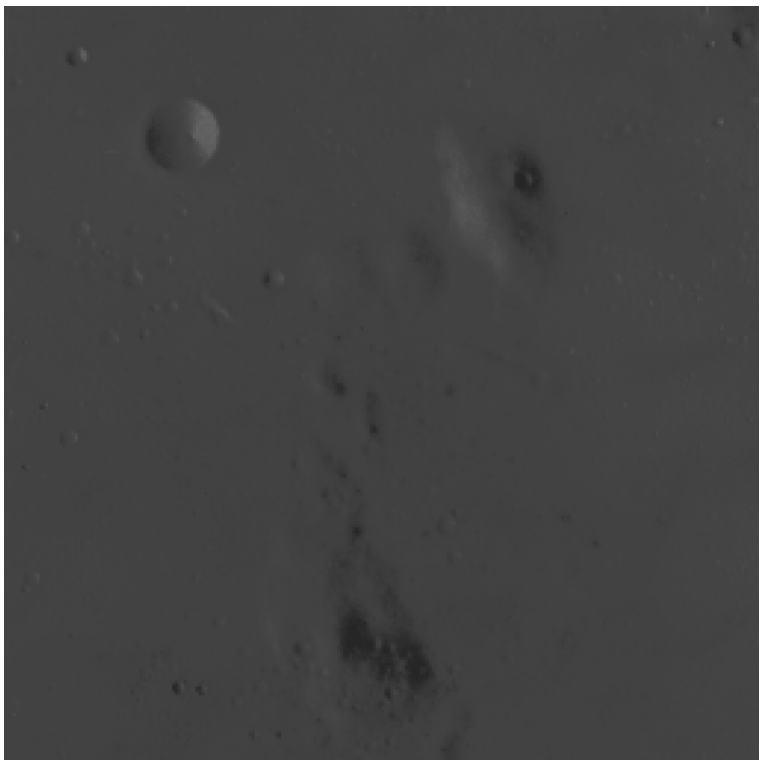
$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p_r(r_j), \qquad k = 0, 1, \ldots, L-1,$$

and round s_k to the nearest integer. Because many input levels can round to the same output level and no level can be split, the discrete result is only *approximately* flat. The output histogram usually shows tall spikes separated by gaps (Figure 3.4, middle). What *is* guaranteed is that the output spans the full range and that the CDF of the output is close to a straight line.

```
def equalize(img, L=256):
    hist = np.bincount(img.ravel(), minlength=L)      # n_k
    cdf = np.cumsum(hist) / img.size                  # sum of p_r(r_j), j <= k
    lut = np.round((L - 1) * cdf).astype(np.uint8)    # s_k
    return lut[img]
```

OpenCV's `cv2.equalizeHist` uses a slightly different normalization, $(\mathrm{cdf}_k - \mathrm{cdf}_{\min})/(MN - \mathrm{cdf}_{\min})$, so the darkest occupied level maps to 0. Results differ by at most a level or so.

Drag the slider to compare a deliberately low-contrast image with its globally equalized version. Notice that equalization is not "free": it exaggerates the large smooth areas and grain, because it spends contrast wherever the pixel *counts* are high, not where the *interesting* detail is.



Input



Equalized

A dim, flat-looking lunar surface image and its histogram-equalized version with strong contrast.

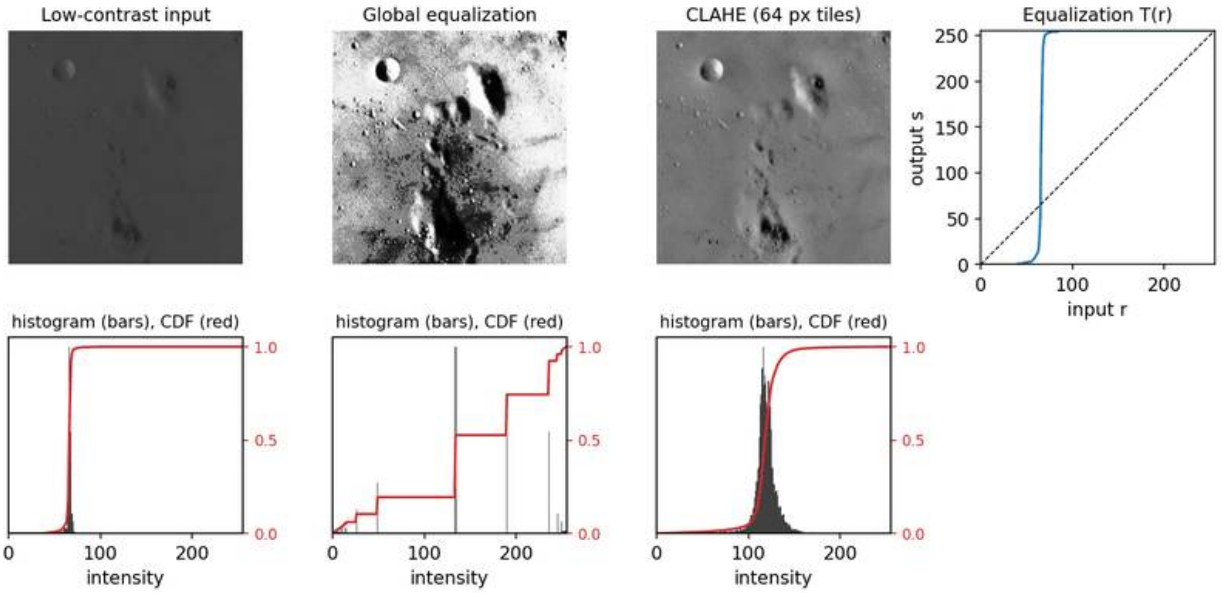


Figure 3.4 — Global histogram equalization versus CLAHE. The input occupies a narrow band of gray levels, so the equalization curve $T(r)$ (right) is almost a step, and the output histogram is spread out with gaps. CLAHE limits the slope and adapts per region.

Histogram matching (specification)

Sometimes a flat histogram is the wrong target. We may want an image to look like a reference image, or to have a histogram shape chosen by hand. **Histogram matching** maps the input so that its histogram approximates a specified PDF $p_z(z)$.

The trick is to use equalization twice. Equalizing the input gives $s = T(r)$ with a uniform s. Equalizing the *target* distribution gives

$$G(z) = (L-1)\int_0^z p_z(t)\,dt,$$

which also yields a uniform variable. Two uniform variables can be matched, so we set $G(z) = T(r)$ and solve:

$$z = G^{-1}(T(r)).$$

Here z is the output intensity, G is the scaled CDF of the desired distribution and t is a dummy variable. In the discrete case G is a staircase and may not be invertible, so for each input level we take the smallest z with $G(z) \geq T(r)$. That is a single `searchsorted` call:

```
def match(img, ref, L=256):
```

```
    cdf_in = np.cumsum(np.bincount(img.ravel(), minlength=L)) / img.size
    cdf_ref = np.cumsum(np.bincount(ref.ravel(), minlength=L)) / ref.size
    # For each input level, the smallest z with G(z) >= T(r).
    lut = np.searchsorted(cdf_ref, cdf_in).clip(0, L - 1).astype(np.uint8)
    return lut[img]
```

`skimage.exposure.match_histograms` implements the same idea, including per-channel matching for color images.

Both equalization and matching are lookup tables, so all pixels with the same input value get the same output value; a large group of equal pixels cannot be split, and the output histogram only approximates the target. Exact histogram specification [5] removes this limit by first putting the pixels in a strict order, breaking ties with local means.

Online only: Exact histogram matching — https://blog.shyandram.dev/en/learn/image-processing/intensity-spatial-filtering/notes/exact-histogram-matching/

Local histogram processing

Global methods use one curve for the whole image. If a dark corner holds the important detail but contains few pixels, the global histogram barely notices it.

Local histogram equalization computes the transformation from the histogram of a neighborhood around each pixel and applies it only to the center pixel, then moves on. **Adaptive histogram equalization** (**AHE**) and its variants were studied systematically by Pizer and colleagues, including the problem that AHE over-amplifies noise in nearly uniform regions, where the local histogram is a single spike and the CDF turns into a near-vertical step [2].

Contrast-limited AHE (**CLAHE**) fixes this by limiting the slope of the mapping. Since the slope of an equalization curve is proportional to the histogram height, clipping each local histogram at a ceiling and redistributing the clipped counts across all bins caps the contrast gain. To make it fast, the image is split into tiles; a mapping is computed per tile, and each pixel's output is interpolated bilinearly from the four nearest tile mappings, which avoids visible tile seams. Zuiderveld's Graphics Gems chapter gave the widely reused implementation [3], and Reza described a real-time hardware realization [4].

```
import cv2
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
out = clahe.apply(data.moon())         # uint8 in, uint8 out
```

In scikit-image, the equivalent is `skimage.exposure.equalize_adapthist(img, clip_limit=0.01)`.

Using histogram statistics for image enhancement

The mean of a histogram measures average brightness, and its standard deviation measures contrast. Computed over a small window around each pixel, they become the **local mean** and the **local standard deviation**, which let us enhance only the regions that need it. A typical rule boosts a pixel only where its neighborhood is dark (a low local mean) and has some, but not too much, contrast (a moderate local standard deviation), which skips both flat background and strong edges. Both local statistics come from two box filters, one applied to I and one to I^2.

Online only: Enhancement with local histogram statistics — https://blog.shyandram.dev/en/learn/image-processing/intensity-spatial-filtering/notes/enhancement-with-local-histogram-statistics/

Fundamentals of spatial filtering

Plain version. A spatial filter slides a small grid of weights (a **kernel**) across the image. At each position, it multiplies the weights by the pixels underneath and adds the products. The result is the new value of the center pixel.

The convolution tutorial covers the mechanics with a from-scratch NumPy loop, OpenCV calls and border handling. Here we focus on the theory the rest of the chapter needs.

Linear spatial filtering and correlation

Using that tutorial's notation, for a $(2k+1) \times (2k+1)$ kernel K the **correlation** of K with I is

$$(K \star I)(x,y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} K(i,j)\, I(x+i,\ y+j),$$

where (i,j) are offsets from the kernel's center and k is the kernel's half-width. Odd sizes give the kernel a well-defined center. The filter is **linear** because the output is a linear combination of input values: filtering $aI_1 + bI_2$ gives a times the result for I_1 plus b times the result for I_2. It is also **shift-invariant** because the same weights are used everywhere.

Correlation versus convolution

Convolution is the same computation with the kernel rotated by 180°:

$$(K * I)(x,y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} K(i,j)\, I(x-i,\ y-j).$$

The two coincide for kernels that are symmetric under a 180° rotation (box, Gaussian, the Laplacian), and differ for antisymmetric ones (Sobel, any directional derivative), where they produce the same response with opposite sign. The useful difference shows up with an **impulse** (a single 1 among zeros): correlating a kernel with an impulse produces a rotated copy of the kernel, while

convolving produces an exact copy. This is why convolution, not correlation, is the operation with clean algebra:

- **commutative:** $K * I = I * K$,
- **associative:** $K_1 * (K_2 * I) = (K_1 * K_2) * I$, so a chain of filters can be merged into one kernel,
- **distributive:** $K * (I_1 + I_2) = K * I_1 + K * I_2$.

Associativity is what the next two subsections rely on. As noted in the convolution tutorial, `cv2.filter2D` computes correlation, and most deep-learning "convolution" layers also compute correlation; since their weights are learned, the flip does not matter there.

Separable kernels

A kernel is **separable** if it is the outer product of a column vector $\mathbf{c}$ and a row vector $\mathbf{r}$:

$$K = \mathbf{c}\,\mathbf{r}^{\top}.$$

Then $K * I = \mathbf{c} * (\mathbf{r}^{\top} * I)$: filter every row with $\mathbf{r}$, then every column with $\mathbf{c}$. For an $m \times n$ kernel this costs $m + n$ multiplications per pixel instead of mn, so the speedup is $mn/(m + n)$, which is 3.5× for a 7×7 kernel and grows linearly with size. A matrix is an outer product exactly when its **rank** is 1, which also gives a test: compute the singular value decomposition and check that only one singular value is nonzero. Box and Gaussian kernels are separable; so is the Sobel kernel, which is a smoothing vector times a differencing vector.

```
g1 = cv2.getGaussianKernel(7, 1.5)          # 7x1 column, sums to 1
K = g1 @ g1.T                               # 7x7 Gaussian kernel
print(np.linalg.matrix_rank(K))             # 1  ->  separable
f = img_as_float(data.camera()).astype(np.float32)
a = cv2.filter2D(f, -1, K.astype(np.float32))
b = cv2.sepFilter2D(f, -1, g1, g1)          # two 1-D passes
print(np.abs(a - b).max())                  # ~1e-7
```

Spatial and frequency domains

The **convolution theorem** states that convolution in space is multiplication in frequency: if $\mathcal{F}$ denotes the Fourier transform, then $\mathcal{F}\{K * I\} = \mathcal{F}\{K\} \cdot \mathcal{F}\{I\}$. So every linear shift-invariant filter has two equivalent descriptions: a kernel K in space, and a **transfer function** $H = \mathcal{F}\{K\}$ that says how much of each frequency survives. Smoothing kernels have transfer functions that are large near zero frequency and small at high frequencies: they are **lowpass**. Sharpening kernels do the opposite: they are **highpass**. Chapter 4 develops this view properly. For now, the vocabulary is enough: "lowpass" means "smooth", "highpass" means "edges and detail".

Small kernels are usually cheaper to apply in space. Very large kernels are often cheaper in frequency via the fast Fourier transform.

Constructing spatial filter kernels

There are three common recipes:

1. **From a mathematical property.** Averaging gives the box kernel. Derivatives give difference kernels (the Laplacian, Sobel).
2. **By sampling a continuous function.** Sample a 2-D Gaussian on the integer grid, then normalize the weights to sum to 1.
3. **From a frequency specification.** Design H in the frequency domain and take its inverse transform to get a spatial kernel, usually truncated to a manageable size.

Two sanity checks apply to all of them. A smoothing kernel's weights should sum to 1, so flat regions keep their brightness. A derivative kernel's weights should sum to 0, so flat regions give 0.

Smoothing (lowpass) spatial filters

Plain version. Replace each pixel by an average of its neighborhood. Random noise is pushed toward the average and fades, but sharp edges are also smeared, because the average straddles both sides.

Box filter

The $m \times n$ box kernel has every weight equal to $1/(mn)$. It is the simplest lowpass filter and is separable, and with a running sum it costs a constant amount of work per pixel regardless of size. Its weakness is its frequency response: the transform of a box is a sinc-like function with side lobes, so it lets some high frequencies through with flipped sign, and it smears in a square, direction-dependent way. Small text and thin lines can turn into blocky streaks.

Gaussian filter

The Gaussian kernel samples

$$w(s,t) = C\, e^{-\frac{s^2+t^2}{2\sigma^2}},$$

where (s,t) are offsets from the kernel's center, σ (the standard deviation) sets the amount of blur, and C is chosen so the weights sum to 1. It has three properties that make it the default smoothing kernel:

- **Isotropy.** It depends only on the distance $\sqrt{s^2+t^2}$, so it blurs equally in every direction.
- **Separability.** $e^{-(s^2+t^2)/2\sigma^2} = e^{-s^2/2\sigma^2}\, e^{-t^2/2\sigma^2}$.
- **A smooth, non-negative frequency response.** Its Fourier transform is also a Gaussian, so there are no side lobes.

Since about 99.7% of a 1-D Gaussian's mass is within $\pm 3\sigma$, a kernel of size about 6σ, rounded up to the next odd integer, captures almost all of it. Larger kernels waste work, smaller ones truncate the tails. Convolving two Gaussians with σ_1 and σ_2 gives a Gaussian with $\sigma = \sqrt{\sigma_1^2 + \sigma_2^2}$, so repeated small blurs compose predictably.

Order-statistic (nonlinear) filters

Order-statistic filters **sort** the values in the neighborhood and pick one. They are nonlinear: no kernel can represent them, and the convolution theorem does not apply.

- **Median filter:** output the middle value (the 50th percentile). It is the standard remedy for **impulse noise** (salt-and-pepper), isolated pixels forced to black or white. An impulse is an extreme value, so it sorts to the end of the list and is almost never the median. As long as fewer than half the pixels in the window are corrupted, the median ignores them. It also preserves step edges far better than a linear average of the same size.
- **Max filter** (100th percentile): finds the brightest point; it removes pepper (dark) noise but grows bright specks.
- **Min filter** (0th percentile): the reverse; removes salt (bright) noise but grows dark specks.

Max and min filters reappear in Chapter 9 as grayscale dilation and erosion.

```
from scipy import ndimage as ndi

f = img_as_float(data.camera()).astype(np.float32)
noisy = f.copy()
u = np.random.default_rng(0).random(f.shape)
noisy[u < 0.05], noisy[u > 0.95] = 0.0, 1.0       # 10% salt-and-pepper

box = cv2.blur(noisy, (5, 5))
gauss = cv2.GaussianBlur(noisy, (0, 0), sigmaX=1.5)
med = cv2.medianBlur(noisy, 3)
mn, mx = ndi.minimum_filter(noisy, 3), ndi.maximum_filter(noisy, 3)
```

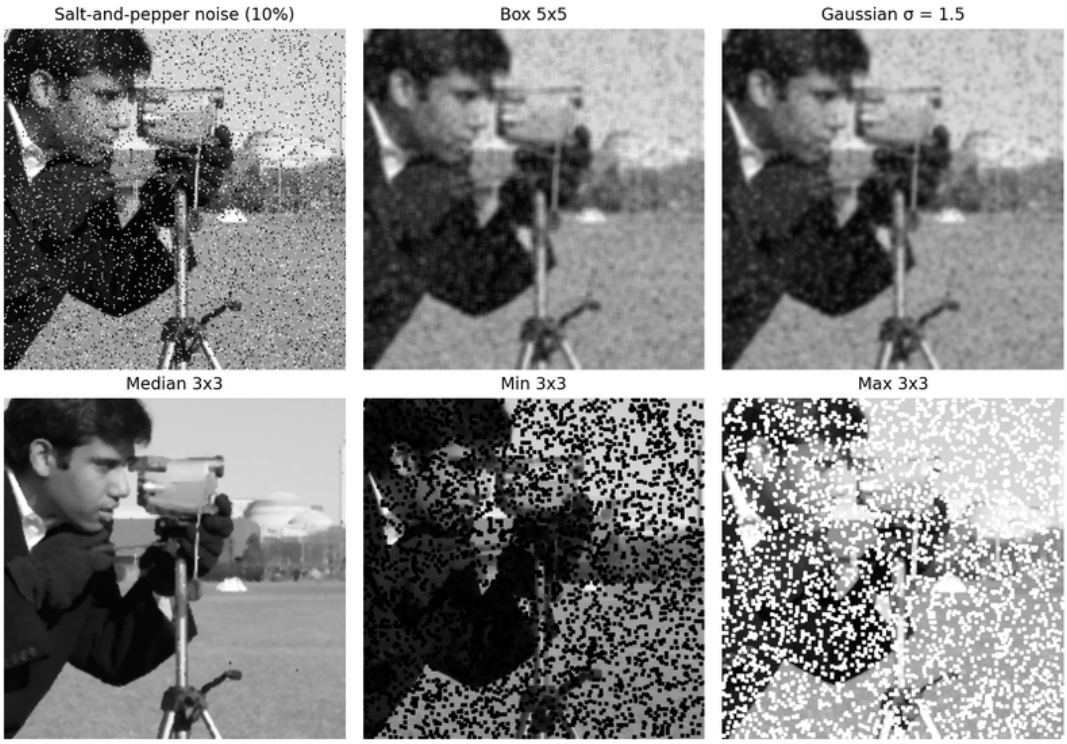


Figure 3.5 — Smoothing 10% salt-and-pepper noise. Linear filters (box, Gaussian) spread each impulse into a gray smudge. The 3×3 median removes nearly all of it. Min and max each remove one polarity and enlarge the other.

Sharpening (highpass) spatial filters

Plain version. Smoothing averages; sharpening does the opposite and highlights *differences*. Differences between neighbors are derivatives, so sharpening is built from derivatives.

First and second derivatives

For a 1-D signal $f(x)$ on a pixel grid, the simplest approximations are

$$\frac{\partial f}{\partial x} = f(x+1) - f(x), \qquad \frac{\partial^2 f}{\partial x^2} = f(x+1) + f(x-1) - 2f(x).$$

Figure 3.6 applies both to a hand-made profile with a ramp, a one-pixel-wide line and a step. Reading it gives the rules that decide which derivative to use:

- In **flat** regions, both derivatives are 0.
- Along a **ramp**, the first derivative is nonzero all the way, which makes thick responses. The second derivative is nonzero only at the ramp's start and end.

- At a **thin line**, the second derivative responds strongly with a double response (positive, strong negative, positive), and its magnitude is larger than the first derivative's. Fine detail is exactly what sharpening should emphasize.
- At a **step**, the second derivative produces a positive and a negative value next to each other. The **zero crossing** between them locates the edge precisely, which Chapter 10 uses for edge detection.

So for enhancing fine detail, the second derivative is the more natural choice.

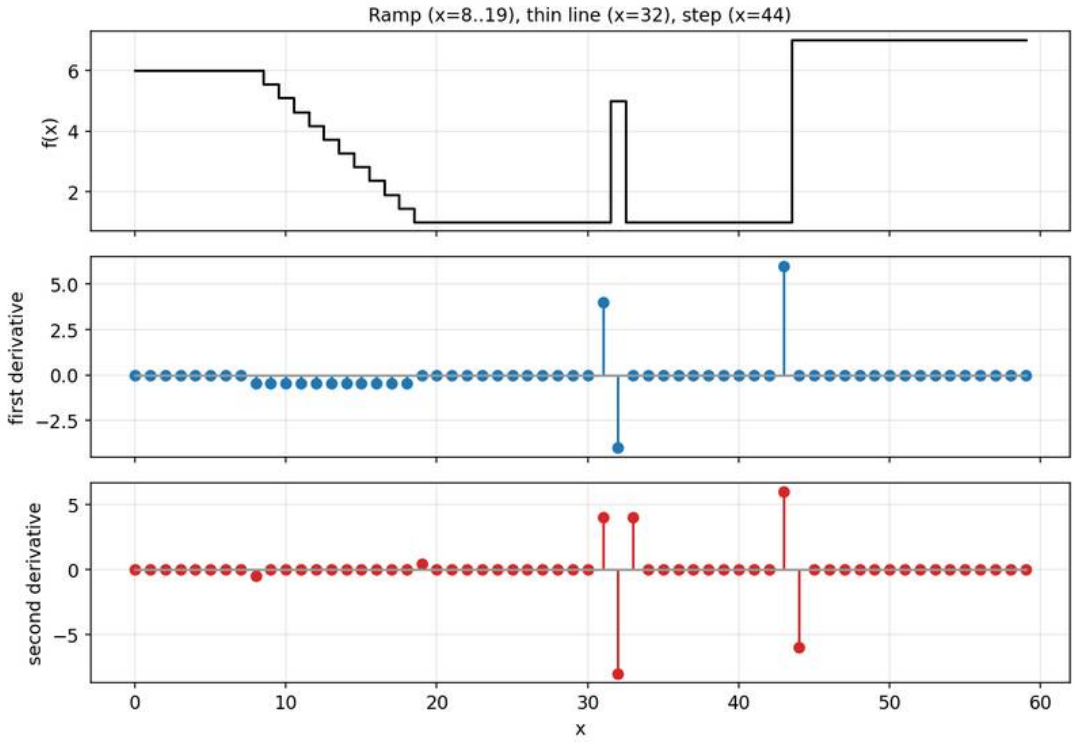


Figure 3.6 — First and second differences of a synthetic profile. The second derivative ignores the inside of the ramp, responds most strongly to the thin line, and changes sign across the step.

The Laplacian

The simplest **isotropic** second-derivative operator (one whose response does not change when the image is rotated) is the **Laplacian**:

$$\nabla^2 I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}.$$

Adding the 1-D difference above in x and in y gives

$$\nabla^2 I(x,y) = I(x+1,y) + I(x-1,y) + I(x,y+1) + I(x,y-1) - 4I(x,y),$$

which is correlation with the kernel

$$K_{\nabla^2} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

Adding the two diagonal directions as well gives a version with 1 in all eight outer cells and -8 in the center, which is isotropic in 45° steps instead of 90° steps. Flipping all the signs gives equally valid kernels. The weights sum to 0, so flat regions vanish and only changes survive.

Sharpening with the Laplacian. The Laplacian image is mostly gray (zero) with bright and dark fringes at edges. Adding it back to the original puts those fringes on the edges:

$$G(x,y) = I(x,y) + c\,\nabla^2 I(x,y),$$

where $c = -1$ for kernels with a negative center (like K_{∇^2} above) and $c = +1$ for kernels with a positive center. The sign matters: with the wrong sign, edges get blurred instead of sharpened. Because convolution is distributive, $I - \nabla^2 I$ is itself a single kernel, with center 5 and -1 above, below and to each side, which is exactly the `sharpen` preset in the playground below.

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

Try the `gaussian` and `box-blur` presets for smoothing, then `laplacian` to see the bare second-derivative response. Changing the center of the sharpen kernel from 5 to 6 makes the weights sum to 2 and brightens the whole image, which shows why the sum matters.

Unsharp masking and highboost filtering

A trick from photographic darkrooms: subtract a blurred copy to isolate the detail, then add the detail back.

$$g_{\text{mask}}(x,y) = I(x,y) - \bar{I}(x,y), \qquad G(x,y) = I(x,y) + \alpha\, g_{\text{mask}}(x,y),$$

where $\bar{I}$ is a blurred (lowpass) version of I, g_{mask} is the **unsharp mask** (the detail the blur removed), and $\alpha \geq 0$ is a weight. With $\alpha = 1$ this is **unsharp masking**. With $\alpha > 1$ it is **highboost filtering**. With $0 < \alpha < 1$ the effect is gentle. Because g_{mask} is a highpass signal, this is another way of adding highpass detail, and if $\bar{I}$ is a Gaussian blur, the blur's σ sets the *size* of the details that get boosted.

Large α causes visible **halos**: overshoot on both sides of strong edges, and the output may leave the valid range, so clip it.

The gradient and Sobel operators

First derivatives are used through the **gradient**, the vector of partial derivatives

$$\nabla I = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \partial I/\partial x \\ \partial I/\partial y \end{bmatrix},$$

which points in the direction of the fastest increase of intensity. Its length, the **gradient magnitude**

$$M(x,y) = \sqrt{g_x^2 + g_y^2} \approx |g_x| + |g_y|,$$

is large at edges and near 0 in flat regions. The right-hand approximation avoids the square root but is not isotropic.

The **Sobel operators** estimate g_x and g_y with 3×3 kernels:

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \qquad K_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

Each is separable: $K_x = [1,2,1]^\top [-1,0,1]$, a central difference in one direction multiplied by a small smoothing in the other. The 2 in the middle gives more weight to the center row or column, which reduces the operator's sensitivity to noise. (The convolution tutorial uses K_x as its edge-detection example.) The gradient magnitude is rarely used to sharpen directly; it is more often used to *find* edges, or as a mask that says where sharpening should be allowed, as in the combined example below.

```
f = img_as_float(data.chelsea()[..., 1]).astype(np.float32)
lap_k = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]], np.float32)
lap = cv2.filter2D(f, -1, lap_k)
sharp = np.clip(f - lap, 0, 1)                # c = -1 because the center is negative

blur = cv2.GaussianBlur(f, (0, 0), 2)
mask = f - blur                               # g_mask
unsharp = np.clip(f + 1.0 * mask, 0, 1)       # alpha = 1
highboost = np.clip(f + 3.0 * mask, 0, 1)     # alpha = 3

gx = cv2.Sobel(f, cv2.CV_32F, 1, 0, ksize=3)
gy = cv2.Sobel(f, cv2.CV_32F, 0, 1, ksize=3)
M = np.hypot(gx, gy)                          # gradient magnitude
```

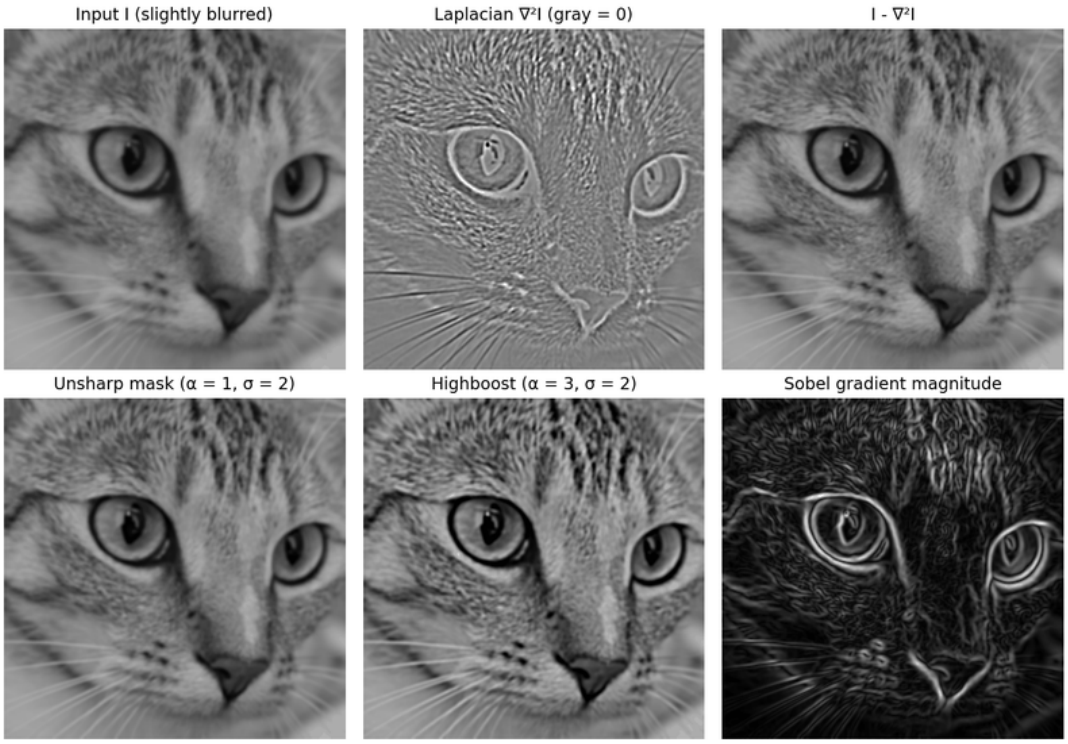


Figure 3.7 — Sharpening a slightly blurred image. Top: input, Laplacian (gray is zero), and $I - \nabla^2 I$. Bottom: unsharp masking, highboost filtering, and the Sobel gradient magnitude.

Always filter in floating point. A `uint8` pipeline clips the negative values of a Laplacian or a Sobel response to 0 and silently loses half the information.

Highpass, bandreject and bandpass filters from lowpass filters

Plain version. If you know how to keep the smooth part, you also know how to keep everything *except* the smooth part: subtract.

Let δ be the **unit impulse kernel**: a 1 at the center and 0 elsewhere, so $\delta * I = I$. Its transfer function equals 1 at every frequency (an all-pass filter). If K_{LP} is a lowpass kernel with transfer function H_{LP}, then

$$K_{\text{HP}} = \delta - K_{\text{LP}} \quad \Longleftrightarrow \quad H_{\text{HP}} = 1 - H_{\text{LP}}.$$

Unsharp masking is exactly this: $g_{\text{mask}} = I - \bar{I} = (\delta - K_{\text{LP}}) * I$.

To isolate a **band** of frequencies, combine two lowpass filters. Let K_1 be a Gaussian with small σ_1 (a high cutoff, so it keeps more detail) and K_2 a Gaussian with larger σ_2 (a low cutoff). Then

$$K_{\text{BP}} = K_1 - K_2, \qquad K_{\text{BR}} = \delta - K_{\text{BP}} = K_2 + (\delta - K_1).$$

The **bandpass** kernel K_{BP} keeps frequencies that K_1 passes but K_2 blocks. The **bandreject** kernel

K_{BR} is the sum of a lowpass and a highpass and removes only that middle band. The bandpass $K_1 - K_2$ is the **difference of Gaussians** (DoG), which is also a good approximation of the Laplacian of a Gaussian and is the basis of the scale-space detectors in Chapter 12. The weights confirm the design: highpass and bandpass kernels sum to 0, and the bandreject kernel sums to 1.

```
def gauss_kernel(sigma):
    n = 2 * int(np.ceil(3 * sigma)) + 1          # about 6 sigma, odd
    g = cv2.getGaussianKernel(n, sigma)
    return g @ g.T

def delta_like(K):
    D = np.zeros_like(K); D[K.shape[0] // 2, K.shape[1] // 2] = 1
    return D

lp2 = gauss_kernel(4.0)                          # low cutoff
lp1 = gauss_kernel(1.0)                          # high cutoff
lp1 = np.pad(lp1, (lp2.shape[0] - lp1.shape[0]) // 2)   # zero-pad to the same size
hp = delta_like(lp1) - lp1                       # highpass, sums to 0
bp = lp1 - lp2                                   # bandpass (DoG), sums to 0
br = delta_like(bp) - bp                         # bandreject, sums to 1
```

Combining spatial enhancement methods

Plain version. Real images rarely have only one problem. A useful enhancement is usually a short pipeline, and the order of the steps matters.

Here is a worked example. The input (Figure 3.8a) is a portrait that is underexposed, has a little Gaussian noise, and has 4% salt-and-pepper impulses. We want a brighter, crisper picture without amplifying the noise.

1. **Remove impulses first, with a 3×3 median filter** (b). Impulses are extreme values; any later contrast boost or sharpening would make them worse, and a linear blur would only spread them. The median removes them while keeping edges.
2. **Lift the shadows with a gamma of 0.45** (c). A point operation is the right tool for a global brightness problem. Doing it after the median means the curve is not distorted by black and white impulses.
3. **Build an edge-weight map** (d). Blur the image with a Gaussian ($\sigma = 2$), compute the Sobel gradient magnitude M of the *blurred* image, and normalize it to $w = \min(M/M_{90}, 1)$, where M_{90} is the 90th percentile of M. Because it is computed after smoothing, w is near 1 on real edges and near 0 on the residual noise in flat areas.
4. **Apply unsharp masking gated by that map** (f): $G = I_c + \alpha\, w \cdot (I_c - \bar{I}_c)$, with I_c the gamma-corrected image, $\bar{I}_c$ its Gaussian blur, $\alpha = 2$, and the product taken pixel by pixel.

Panel (e) shows plain unsharp masking with the same α: the background and the dark collar get grainy, because the noise is high-frequency detail too. The gated version (f) sharpens the face, hair and the suit's outlines nearly as much, while leaving the smooth background clean. This pattern,

using a smoothed first-derivative image to *decide where* a second-order sharpening is applied, is a hand-made version of the edge-aware processing that the Modern view describes.



Figure 3.8 — A four-step enhancement pipeline: (a) input, (b) median 3×3, (c) gamma 0.45, (d) edge weight from a smoothed Sobel magnitude, (e) plain unsharp masking for comparison, (f) unsharp masking gated by (d).

Each step uses one idea from this chapter. The order follows three rules of thumb: remove outliers before any step that amplifies differences, fix global tone with a point operation, and restrict sharpening to where the signal is stronger than the noise.

Modern view

The techniques in this chapter are decades old, and they are still everywhere: CLAHE is a standard preprocessing step in medical imaging and remote sensing, gamma and tone curves are in every camera pipeline, and Gaussian, median and Sobel filters are in every vision library. Research since has pushed in three directions: smarter contrast enhancement, filters that smooth without blurring edges, and learned replacements for both. The surveys below are good entry points.

Contrast enhancement and histogram methods. Pizer et al. [2] remain the reference analysis of adaptive histogram equalization, Zuiderveld's chapter [3] is the compact CLAHE implementation that libraries follow, and Coltuc, Bolon and Chassery [5] show how to reach an *exact* target histogram. For a broader map, Qi et al. [11] survey two decades of image-enhancement methods,

grouped into supervised and unsupervised algorithms, with quality evaluation as a third pillar.

Edge-preserving filters. Box and Gaussian filters average across edges. The **bilateral filter** [6] adds a second weight that decays with intensity difference, so pixels on the other side of an edge contribute little. The **guided filter** [8] models the output as a local linear function of a guide image, needs only box filters, and behaves better near strong edges. Paris et al. [7] survey the bilateral filter and its uses, such as splitting an image into a base layer and a detail layer for tone mapping; Milanfar [9] unifies these filters as data-adaptive weighted averages; and Gingichashvili and Lischinski [10] propose a common methodology for comparing them.

Online only: The guided filter: smoothing that follows a guide image — https://blog.shyandram.dev/en/learn/image-processing/intensity-spatial-filtering/notes/the-guided-filter-smoothing-that-follows-a-guide-image/; The bilateral filter: averaging only with pixels that look alike — https://blog.shyandram.dev/en/learn/image-processing/intensity-spatial-filtering/notes/the-bilateral-filter-averaging-only-with-pixels-that-look-alike/

Online only: Bilateral filtering: average nearby pixels without crossing a strong edge — https://blog.shyandram.dev/en/learn/image-processing/intensity-spatial-filtering/papers/bilateral-filtering-average-nearby-pixels-without-crossing-a-strong-edge/; Guided Filter: fit local linear models instead of comparing every pair — https://blog.shyandram.dev/en/learn/image-processing/intensity-spatial-filtering/papers/guided-filter-fit-local-linear-models-instead-of-comparing-every-pair/

How deep learning changed things. The most interesting finding is how much of this chapter survives *inside* learned systems. Many successful enhancement networks do not output pixels directly; they predict the parameters of a classical operator and then apply it.

- **Learned curves.** Zero-DCE [13] uses a small CNN to predict a brightening curve for every pixel, and it is trained without reference images: the gamma and contrast-stretching idea, with the curve chosen per pixel.
- **Learned edge-aware filters.** HDRNet [15] predicts local affine color transforms in a low-resolution bilateral grid and applies them at full resolution, which turns the bilateral filter's edge-awareness into a learned layer.
- **Low-light enhancement.** The survey by Li et al. [12] reviews deep low-light image and video enhancement by learning strategy, network structure, loss functions and training data.

Online only: The Modern view in more detail — https://blog.shyandram.dev/en/learn/image-processing/intensity-spatial-filtering/papers/the-modern-view-in-more-detail/

What did *not* change: the median filter is still the first answer to impulse noise, the Sobel kernels still appear in loss functions and edge maps, CLAHE is still a default preprocessing step, and gamma handling remains a source of bugs in learned pipelines that mix linear and encoded pixel values. Deep models also tend to inherit the classical failure modes: halos, noise amplification in dark regions and over-enhancement are still the artifacts that reviewers look for.

Key takeaways

- An intensity transformation $s = T(r)$ is a lookup table; its slope is the local contrast gain. Log and $\gamma < 1$ curves expand the shadows, $\gamma > 1$ expands the highlights, and piecewise-linear curves give precise control.
- Histogram equalization uses the scaled CDF as T, which makes the output PDF uniform in the continuous case and approximately flat in the discrete case. Matching chains one equalization with the inverse of another; exact matching first breaks ties with local means.
- Local methods (AHE, CLAHE, local mean and variance rules) adapt to regions; CLAHE limits the contrast gain by clipping each local histogram.
- Linear spatial filtering is correlation with a kernel; convolution flips the kernel and has clean algebra. Separable kernels cost $m + n$ instead of mn operations per pixel.
- Lowpass filters (box, Gaussian) average away noise and blur edges; the nonlinear median filter is the tool for impulse noise.
- Sharpening adds highpass detail: the Laplacian (second derivative), unsharp masking and highboost, while the gradient and Sobel operators measure edge strength.
- Highpass, bandpass and bandreject kernels follow from lowpass ones by subtraction from the impulse δ; good pipelines order steps so that nothing amplifies what an earlier step should have removed.

Exercises

1. **A curve for a foggy photo.** An 8-bit image has all its pixels between 90 and 160. Write the piecewise-linear transformation that maps this range onto $[0, 255]$, give its slope, and say what happens to two neighboring pixels with values 120 and 122.

   Hint

   $T(r) = 255\,(r - 90)/70$ for $90 \le r \le 160$, clipped to $[0, 255]$ outside. The slope is $255/70 \approx 3.64$, so a difference of 2 becomes about 7.3 gray levels. After rounding, $T(120) = 109$ and $T(122) = 117$.

2. **Equalize by hand.** A tiny 3-bit image ($L = 8$) with 16 pixels has counts $n = [0, 8, 4, 2, 2, 0, 0, 0]$ for levels 0 to 7. Compute the equalization lookup table and the output histogram. Why is the output not flat?

   Hint

   The CDF is $[0, 0.5, 0.75, 0.875, 1, 1, 1, 1]$, so $s_k = \text{round}(7 \cdot \text{CDF})$ gives $[0, 4, 5, 6, 7, 7, 7, 7]$ (with 3.5 rounded to 4). The output histogram has 8 pixels at level 4, 4 at 5, 2 at 6 and 2 at 7. The 8 pixels at level 1 share one input value, and a lookup table cannot split them. Only exact histogram specification could.

3. **Does order matter?** Is applying a gamma curve and then a box (averaging) filter the same as applying the box filter and then the gamma curve? Prove your answer or give a counterexample using a 2-pixel average.

Hint

No. Take a 1-D average of two pixels, 0 and 1, with $\gamma = 2$. Gamma first: $(0 + 1)/2 = 0.5$. Box first: $0.5^2 = 0.25$. A nonlinear point operation does not commute with a linear filter. This is also why blurring gamma-encoded images is not the same as blurring linear light.

4. **Separable or not?** Decide whether each kernel is separable, and if so, give the two vectors: (a) the 3×3 Laplacian with -4 in the center; (b) the 3×3 kernel with all entries 1; (c) $\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$.

Hint

(a) No: its rank is 2 (but it is the *sum* of two separable kernels, one per axis). (b) Yes: $[1,1,1]^\top [1,1,1]$. (c) Yes: $[1,2,1]^\top [1,2,1]$, a small binomial approximation of a Gaussian. Check with `np.linalg.matrix_rank`.

5. **Median versus mean on an edge.** A 1-D signal is $[10, 10, 10, 10, 50, 50, 50, 50]$, a perfect step. Apply a 3-sample mean and a 3-sample median (ignore the end samples). Then replace the sample at position 1 (counting from 0) with an impulse of value 255 and repeat. What do you conclude?

Hint

On the clean step, the mean gives a ramp $(\ldots, 10, 23.3, 36.7, 50, \ldots)$ while the median keeps the step exactly. With the impulse, the mean turns it into two raised values of about 92 at positions 1 and 2; the median outputs 10 there and removes it completely.

6. **Design a bandpass kernel.** Using Gaussians with $\sigma_1 = 1$ and $\sigma_2 = 3$, build a bandpass kernel and a bandreject kernel in NumPy. Verify the sums of their weights, filter `skimage.data.brick()` with both, and describe which structures each output keeps.

Hint

Make both kernels the same size (pad the smaller one with zeros), then use $K_1 - K_2$ and $\delta - (K_1 - K_2)$. Sums should be about 0 and 1. The bandpass output keeps the mortar lines and the brick-scale texture, removing both the overall shading and the finest grain; the bandreject output keeps shading and fine grain but weakens the brick pattern.

References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018, Ch. 3. publisher page

2. S. M. Pizer, E. P. Amburn, J. D. Austin, R. Cromartie, A. Geselowitz, T. Greer, B. ter Haar Romeny, J. B. Zimmerman and K. Zuiderveld, "Adaptive histogram equalization and its variations," *Computer Vision, Graphics, and Image Processing*, 1987. doi
3. K. Zuiderveld, "Contrast limited adaptive histogram equalization," in P. S. Heckbert (ed.), *Graphics Gems IV*, Morgan Kaufmann, 1994, pp. 474–485. entry
4. A. M. Reza, "Realization of the contrast limited adaptive histogram equalization (CLAHE) for real-time image enhancement," *Journal of VLSI Signal Processing*, 2004. doi
5. D. Coltuc, P. Bolon and J.-M. Chassery, "Exact histogram specification," *IEEE Transactions on Image Processing*, 2006. doi
6. C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Proc. IEEE International Conference on Computer Vision (ICCV)*, 1998. pdf
7. S. Paris, P. Kornprobst, J. Tumblin and F. Durand, "Bilateral filtering: Theory and applications," *Foundations and Trends in Computer Graphics and Vision*, vol. 4, no. 1, pp. 1–73, 2009. doi
8. K. He, J. Sun and X. Tang, "Guided image filtering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(6):1397–1409, 2013 (first presented at ECCV 2010). doi
9. P. Milanfar, "A tour of modern image filtering," *IEEE Signal Processing Magazine*, vol. 30, no. 1, pp. 106–128, 2013. IEEE Xplore
10. S. Gingichashvili and D. Lischinski, "Evaluation and comparison of edge-preserving filters," arXiv:2012.13778, 2020. arXiv
11. Y. Qi, Z. Yang, W. Sun et al., "A comprehensive overview of image enhancement techniques," *Archives of Computational Methods in Engineering*, 2021. doi
12. C. Li, C. Guo, L. Han, J. Jiang, M.-M. Cheng, J. Gu and C. C. Loy, "Low-light image and video enhancement using deep learning: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021 (arXiv:2104.10729). arXiv
13. C. Guo, C. Li, J. Guo, C. C. Loy, J. Hou, S. Kwong and R. Cong, "Zero-reference deep curve estimation for low-light image enhancement," in *Proc. IEEE/CVF CVPR*, 2020. arXiv
14. H. Zeng, J. Cai, L. Li, Z. Cao and L. Zhang, "Learning image-adaptive 3D lookup tables for high performance photo enhancement in real-time," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020. arXiv
15. M. Gharbi, J. Chen, J. T. Barron, S. W. Hasinoff and F. Durand, "Deep bilateral learning for real-time image enhancement," *ACM Transactions on Graphics (SIGGRAPH)*, 2017. arXiv
16. H. Wu, S. Zheng, J. Zhang and K. Huang, "Fast end-to-end trainable guided filter," in *Proc. IEEE/CVF CVPR*, 2018. arXiv
17. Y. Cai, H. Bian, J. Lin, H. Wang, R. Timofte and Y. Zhang, "Retinexformer: One-stage Retinex-based transformer for low-light image enhancement," in *Proc. IEEE/CVF ICCV*, 2023. arXiv
18. S. Paris and F. Durand, "A fast approximation of the bilateral filter using a signal processing approach," *International Journal of Computer Vision*, vol. 81, no. 1, pp. 24–52, 2009 (first presented at ECCV 2006). doi

IP 04 • Filtering in the Frequency Domain

How the Fourier transform turns an image into a recipe of waves, and how editing that recipe smooths, sharpens and removes periodic noise.

Online edition: https://blog.shyandram.dev/en/learn/image-processing/frequency-domain/

What you'll learn

- What the Fourier transform says about a signal, and why "frequency" is a useful way to describe an image
- How sampling works, what the Nyquist rate is, and why too few samples cause aliasing and moiré
- The discrete Fourier transform (DFT) in one and two dimensions, and the properties you use every day: centering, symmetry, spectrum and phase, and the convolution theorem
- The step-by-step recipe for filtering in the frequency domain, including the padding that prevents wraparound error
- Lowpass, highpass, homomorphic, bandreject and notch filters, and why the "ideal" filter rings
- How the fast Fourier transform (FFT) makes all of this cheap, and where Fourier ideas show up in modern deep learning

The big picture

In Chapter 3 we filtered images by sliding a small kernel over the pixels. This chapter looks at the same image from a completely different angle. Instead of asking "what is the brightness at this pixel?", we ask "how much of each kind of wave does this image contain?" That second description is called the image's **spectrum**, and the tool that produces it is the **Fourier transform**.

> **Explain like I'm five**
>
> Think of a song played by a band. Your ear hears one mixed sound, but a musician can tell you it is "a low bass note, a middle piano chord and a high cymbal." The Fourier transform does the same thing for a picture: it says how much slow change (big smooth areas, like the bass), and how much fast change (sharp edges and fine texture, like the cymbal) is inside. Once you know the mix, you can turn the cymbal down to make the picture smoother, or turn it up to make it crisper, and then play the song again.

Why bother? Three reasons. First, some operations are much easier to describe as "keep these frequencies, remove those" than as a kernel. Second, by the convolution theorem, a large convolution becomes a simple multiplication, which the FFT makes fast. Third, the frequency view explains things that are mysterious in the pixel view: why shrinking an image creates strange

patterns, why a sharp cutoff filter produces ripples, and why a repeating stripe of noise can be removed almost perfectly. The chapter follows the arrangement of Gonzalez and Woods [1].

Background

Plain version. Any reasonable signal can be built by adding up sine and cosine waves of different frequencies, each with its own strength and starting point. The Fourier transform finds those strengths and starting points.

In the early 1800s Joseph Fourier claimed that a periodic function can be written as a weighted sum of sines and cosines, and that even non-periodic functions with finite area under the curve can be written as an integral of sines and cosines. The striking part is that nothing is lost: you can go forward to the frequency description and back again exactly.

For images this matters in two ways. A **low frequency** is a wave that changes slowly across the image: it carries overall brightness and large smooth regions. A **high frequency** changes quickly: it carries edges, fine texture and noise. Smoothing an image means weakening the high frequencies. Sharpening means strengthening them. This chapter makes that statement precise.

Fourier analysis only became a practical image-processing tool after the fast Fourier transform appeared in 1965 [4]. Before that, computing a transform of an image was far too slow.

Preliminary concepts

Plain version. Before we can transform images, we need five tools: complex numbers (to store a wave's strength and starting point together), Fourier series (sums of waves), impulses (perfect "spikes"), the Fourier transform itself, and convolution.

Complex numbers

A complex number is $C = R + jI$, where R is the real part, I is the imaginary part, and $j = \sqrt{-1}$. It can also be written in polar form

$$C = |C|\, e^{j\theta}, \qquad |C| = \sqrt{R^2 + I^2}, \qquad \theta = \operatorname{atan2}(I, R)$$

where $|C|$ is the magnitude (length) and θ is the angle. Euler's formula $e^{j\theta} = \cos\theta + j\sin\theta$ links the two forms. This is why the Fourier transform uses complex exponentials: one complex number holds both the **amplitude** of a wave and its **phase** (where it starts). The complex conjugate is $C^* = R - jI$.

Fourier series

A function $f(t)$ that repeats with period T can be written as

$$f(t) = \sum_{n=-\infty}^{\infty} c_n \, e^{j\frac{2\pi n}{T}t}, \qquad c_n = \frac{1}{T}\int_{-T/2}^{T/2} f(t)\, e^{-j\frac{2\pi n}{T}t}\, dt$$

where n is an integer, c_n is the (complex) coefficient of the wave that completes n cycles per period, and t is the continuous variable (time or position).

Impulses and the sifting property

A continuous **unit impulse** $\delta(t)$ is an idealized spike: zero everywhere except at $t = 0$, with total area 1, so $\int_{-\infty}^{\infty} \delta(t)\, dt = 1$. Its most useful property is **sifting**:

$$\int_{-\infty}^{\infty} f(t)\, \delta(t - t_0)\, dt = f(t_0)$$

where t_0 is any location. Multiplying by a shifted impulse and integrating "picks out" the value of f at t_0. The discrete impulse $\delta(x)$ is 1 at $x = 0$ and 0 elsewhere, and sifting becomes $\sum_x f(x)\, \delta(x - x_0) = f(x_0)$.

An **impulse train** $s_{\Delta T}(t) = \sum_{n=-\infty}^{\infty} \delta(t - n\Delta T)$ is a row of impulses spaced ΔT apart. It is the mathematical model of sampling.

The Fourier transform of a continuous function

The Fourier transform of $f(t)$ and its inverse are

$$F(\mu) = \int_{-\infty}^{\infty} f(t)\, e^{-j2\pi\mu t}\, dt, \qquad f(t) = \int_{-\infty}^{\infty} F(\mu)\, e^{j2\pi\mu t}\, d\mu$$

where μ is the continuous frequency (cycles per unit of t) and $F(\mu)$ is complex in general. Together the two equations are called a **Fourier transform pair**, written $f(t) \Leftrightarrow F(\mu)$.

A worked example builds intuition. Take a box of height A that is nonzero for $|t| \le W/2$. Its transform is

$$F(\mu) = AW\,\frac{\sin(\pi\mu W)}{\pi\mu W} = AW\ \mathrm{sinc}(\mu W)$$

where $\mathrm{sinc}(m) = \sin(\pi m)/(\pi m)$. Two lessons follow. A function with sharp edges has a transform that spreads out with slowly decaying ripples. And there is an inverse relationship: a wider box gives a narrower sinc. Narrow in one domain means wide in the other. We will meet both lessons again when the "ideal" filter rings.

Two more pairs are worth memorizing. The transform of an impulse is a constant: $\delta(t) \Leftrightarrow 1$. And the transform of an impulse train with spacing ΔT is another impulse train with spacing $1/\Delta T$:

$$s_{\Delta T}(t) \Leftrightarrow S(\mu) = \frac{1}{\Delta T}\sum_{n=-\infty}^{\infty} \delta\!\left(\mu - \frac{n}{\Delta T}\right)$$

Convolution

The convolution of two continuous functions is

$$(f \star h)(t) = \int_{-\infty}^{\infty} f(\tau)\, h(t-\tau)\, d\tau$$

where τ is a dummy integration variable. The **convolution theorem** says

$$(f \star h)(t) \Leftrightarrow H(\mu)F(\mu), \qquad f(t)\, h(t) \Leftrightarrow (H \star F)(\mu)$$

Convolution in one domain is multiplication in the other. This single fact is the reason frequency-domain filtering exists.

Sampling and the Fourier transform of sampled functions

Plain version. A camera does not record a continuous scene; it measures it at a grid of points. If the points are too far apart, fast details get confused with slow ones and you see patterns that are not really there. Sample at more than twice the fastest frequency and you can, in principle, rebuild the original exactly.

Sampling as multiplication

Model sampling as multiplying $f(t)$ by an impulse train:

$$\tilde{f}(t) = f(t)\, s_{\Delta T}(t) = \sum_{n=-\infty}^{\infty} f(n\Delta T)\, \delta(t - n\Delta T)$$

where ΔT is the sampling interval and $f(n\Delta T)$ are the samples. By the convolution theorem, multiplication becomes convolution with $S(\mu)$, and convolving with a train of impulses just copies the spectrum:

$$\tilde{F}(\mu) = \frac{1}{\Delta T} \sum_{n=-\infty}^{\infty} F\left(\mu - \frac{n}{\Delta T}\right)$$

So **the spectrum of a sampled function is an infinite, periodic sequence of copies of the original spectrum**, spaced $1/\Delta T$ apart.

The sampling theorem and the Nyquist rate

Suppose f is **band-limited**: $F(\mu) = 0$ for $|\mu| > \mu_{\max}$. The copies do not overlap as long as their spacing exceeds the width of one copy:

$$\frac{1}{\Delta T} > 2\mu_{\max}$$

This is the **sampling theorem**, made famous in engineering by Shannon [2]. The quantity $2\mu_{\max}$ is the **Nyquist rate**. Sampling faster than it means the samples contain all the information in f.

Reconstruction

If the copies do not overlap, multiplying $\tilde{F}(\mu)$ by a box $H(\mu)$ of height ΔT that covers $[-\mu_{\max}, \mu_{\max}]$ isolates the original spectrum. In the time domain this is convolution with a sinc, giving the interpolation formula

$$f(t) = \sum_{n=-\infty}^{\infty} f(n\Delta T)\ \operatorname{sinc}\left[\frac{t - n\Delta T}{\Delta T}\right]$$

Every sample contributes a sinc centered on itself, and the sum passes through all samples exactly. Unser's review [3] explains how modern practice replaces the impractical infinite sinc with splines and other compact kernels that approximate it well.

Aliasing

If $1/\Delta T \le 2\mu_{\max}$, neighboring copies overlap and their tails add together. No filter can separate them afterwards: high frequencies now masquerade as low ones. This is **aliasing**. The only cure is prevention: remove the high frequencies with a lowpass (**anti-aliasing**) filter *before* sampling.

A one-line experiment shows the disguise. Sample a 9 Hz cosine at 10 samples per second, where the Nyquist limit is 5 Hz:

```
import numpy as np

fs = 10.0                       # samples per second
t = np.arange(0, 2, 1 / fs)     # 2 seconds of samples
x = np.cos(2 * np.pi * 9 * t)  # a 9 Hz tone: above fs/2 = 5 Hz
X = np.abs(np.fft.rfft(x))
freqs = np.fft.rfftfreq(len(x), d=1 / fs)
print("peak at", freqs[X.argmax()], "Hz")   # -> 1.0 Hz, not 9 Hz
```

The 9 Hz tone is indistinguishable from a 1 Hz tone at these sample times. This is the same effect that makes wagon wheels appear to spin backwards in films.

The discrete Fourier transform of one variable

Plain version. A computer only has a finite list of samples, so it uses a finite version of the Fourier transform. The DFT takes M numbers and returns M numbers, each saying how much of one particular wave is present.

Definition

For samples $f(0), f(1), \ldots, f(M-1)$:

$$F(u) = \sum_{x=0}^{M-1} f(x)\, e^{-j2\pi ux/M}, \qquad f(x) = \frac{1}{M} \sum_{u=0}^{M-1} F(u)\, e^{j2\pi ux/M}$$

where $x = 0, \ldots, M-1$ indexes samples, $u = 0, \ldots, M-1$ indexes discrete frequencies (frequency u completes u cycles over the M samples), and the factor $1/M$ sits on the inverse (other conventions split it as $1/\sqrt{M}$ on each side).

Relation to sampling

Where does this come from? Take the transform of the sampled function $\tilde{F}(\mu)$, which is continuous and periodic with period $1/\Delta T$. Sample one period of it at M equally spaced points, $\mu = m/(M\Delta T)$ for $m = 0, \ldots, M-1$. The result is exactly the DFT above. So the DFT is **samples of one period of the transform of the sampled data**. The spacing between DFT frequencies is

$$\Delta u = \frac{1}{M\Delta T}$$

where $M\Delta T$ is the total duration covered by the samples. Longer recordings give finer frequency resolution.

Two consequences matter in practice. Because we sampled in frequency as well, the DFT treats the data as if it repeats with period M. And both $F(u)$ and $f(x)$ are periodic: $F(u) = F(u+M)$ and $f(x) = f(x+M)$.

The definition is just a matrix-vector product, which is easy to check:

```
import numpy as np

def dft(f):
    M = len(f)
    x = np.arange(M)
    u = x.reshape(-1, 1)
    W = np.exp(-2j * np.pi * u * x / M)   # M x M matrix of complex exponentials
    return W @ f

f = np.random.default_rng(0).standard_normal(64)
print(np.allclose(dft(f), np.fft.fft(f)))  # True
```

Extension to functions of two variables

Plain version. Images have two directions, so the waves have two frequencies: one across and one down. A 2-D wave looks like a set of parallel stripes; its two frequencies set how close the stripes are and which way they point.

The 2-D impulse and continuous transform

The 2-D impulse $\delta(t, z)$ has unit volume and sifts like its 1-D cousin: $\iint f(t, z)\, \delta(t - t_0, z - z_0)\, dt\, dz = f(t_0, z_0)$. The 2-D continuous transform is

$$F(\mu, \nu) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(t, z)\, e^{-j2\pi(\mu t + \nu z)}\, dt\, dz$$

where t, z are the spatial coordinates and μ, ν are the corresponding frequencies.

2-D sampling and aliasing

Sampling on a grid with spacings ΔT and ΔZ replicates the spectrum on a 2-D lattice. A band-limited image (zero outside $|\mu| \le \mu_{\max}, |\nu| \le \nu_{\max}$) is recoverable when

$$\frac{1}{\Delta T} > 2\mu_{\max}, \qquad \frac{1}{\Delta Z} > 2\nu_{\max}$$

Images break this rule constantly. Real scenes contain sharp edges, which are not band-limited at all, so every digital image has some aliasing. It becomes visible when you **downsample** (shrink) an image by dropping pixels. Figure 4.1 uses a zone plate, a pattern whose frequency increases with distance from the center. Keeping every 4th pixel turns the fine outer rings into false circles that were never in the original. Blurring first removes the frequencies that cannot survive, and the false rings disappear.

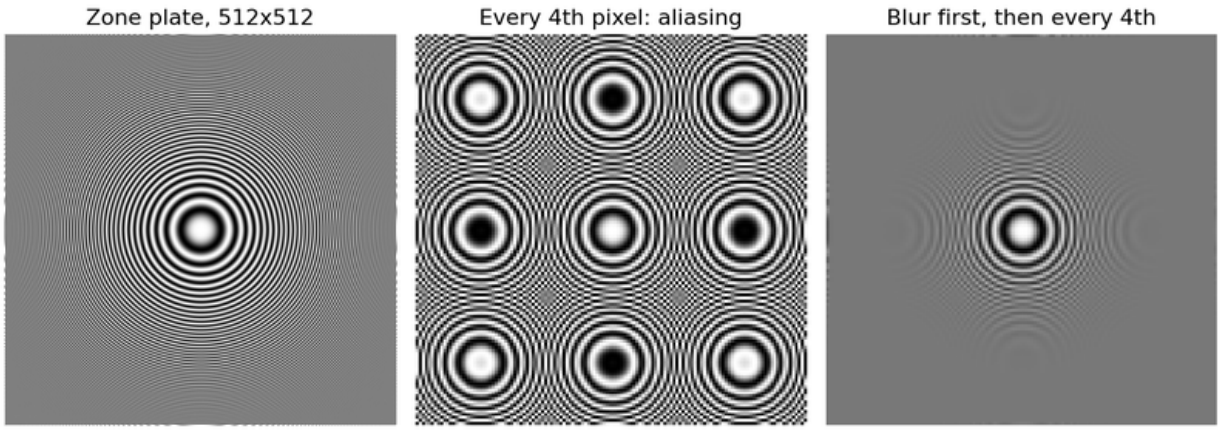


Figure 4.1 — Aliasing in 2-D. Left: a zone plate whose frequency grows outward. Middle: keeping every 4th pixel produces false rings (aliases). Right: a Gaussian blur before subsampling removes the unrepresentable frequencies, leaving a flat gray where the detail was.

Moiré patterns

A **moiré pattern** is aliasing you can see in everyday life: when a fine periodic texture (a shirt's weave, a screen door, the halftone dots of a printed photo) is sampled by a sensor grid of similar

spacing, the two periodicities beat against each other and produce large, slow, wavy bands. The middle panel of Figure 4.1 is a moiré pattern of exactly this kind. Cameras reduce it with an optical anti-aliasing filter in front of the sensor; software reduces it by blurring before resizing.

The 2-D DFT

For an $M \times N$ image $f(x, y)$:

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y)\, e^{-j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}, \qquad f(x,y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v)\, e^{j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}$$

where x, y are pixel coordinates (rows, columns), u, v are discrete frequency indices, and $F(u, v)$ is an $M \times N$ array of complex numbers. Every pixel of F describes one 2-D wave covering the whole image.

Properties of the 2-D DFT

Plain version. The spectrum behaves in predictable ways. Moving an object does not change the spectrum's magnitude. Rotating the object rotates the spectrum. The center of the (shifted) spectrum holds the average brightness. And the most surprising fact: most of what we recognize in a picture is stored in the *phase*, not the magnitude.

Spatial and frequency intervals

If the image was sampled with spacings ΔT and ΔZ, the frequency spacings of the DFT are

$$\Delta u = \frac{1}{M\Delta T}, \qquad \Delta v = \frac{1}{N\Delta Z}$$

A larger field of view gives finer frequency resolution; finer pixels extend the highest frequency the DFT can represent.

Translation and rotation

Shifting the image multiplies the transform by a linear phase, and vice versa:

$$f(x - x_0, y - y_0) \Leftrightarrow F(u,v)\, e^{-j2\pi\left(\frac{ux_0}{M}+\frac{vy_0}{N}\right)}$$

where (x_0, y_0) is the shift. Because $|e^{-j\cdot}| = 1$, **the magnitude $|F|$ is unchanged by translation**; only the phase changes. With polar coordinates $x = r\cos\theta$, $y = r\sin\theta$ and $u = \omega\cos\varphi$, $v = \omega\sin\varphi$, rotating f by θ_0 rotates F by the same angle: $f(r, \theta + \theta_0) \Leftrightarrow F(\omega, \varphi + \theta_0)$.

Periodicity and centering

The DFT and its inverse are periodic in both directions: $F(u,v) = F(u + k_1 M, v + k_2 N)$ for any integers k_1, k_2. A raw DFT array therefore has its zero frequency in the top-left corner, with four quarter-periods meeting at the center. For viewing and for designing filters it is much easier to move zero frequency to the center. Using the translation property with $x_0 = M/2$, $y_0 = N/2$:

$$f(x,y)\,(-1)^{x+y} \Leftrightarrow F(u - M/2,\ v - N/2)$$

which holds for even M and N. In NumPy, `np.fft.fftshift` does the same reordering after the transform.

Symmetry

For a real image, the transform is **conjugate symmetric**: $F(-u,-v) = F^*(u,v)$. Its magnitude is therefore even, $|F(-u,-v)| = |F(u,v)|$, and its phase is odd. Every spike in a spectrum comes with a mirror twin through the center. You will use this when designing notch filters.

Fourier spectrum and phase angle

Write $F(u,v) = R(u,v) + jI(u,v)$. Then

$$|F(u,v)| = \sqrt{R^2 + I^2}, \qquad \phi(u,v) = \operatorname{atan2}(I(u,v), R(u,v)), \qquad P(u,v) = |F(u,v)|^2$$

are the **Fourier spectrum** (magnitude), the **phase angle** and the **power spectrum**. The zero-frequency term is the **DC component**, $F(0,0) = MN\bar{f}$, where $\bar{f}$ is the mean intensity. Because $F(0,0)$ is usually thousands of times larger than everything else, spectra are displayed as $\log(1 + |F|)$.

```
import numpy as np
from skimage import data, img_as_float

f = img_as_float(data.camera())
F = np.fft.fft2(f)
Fc = np.fft.fftshift(F)                       # move (0,0) to the centre
spectrum = np.log1p(np.abs(Fc))               # log(1+|F|) for display
phase = np.angle(Fc)                         # atan2(imaginary part, real part)
print(np.angle(np.array([1+1j, -1+1j]), deg=True))  # [ 45. 135.]

M, N = f.shape
y, x = np.mgrid[0:M, 0:N]
print(np.allclose(np.fft.fft2(f * (-1.0) ** (x + y)), Fc))  # True for even M, N
print(np.isclose(F[0, 0].real, f.sum()))                    # DC term = sum of pixels
```

Figure 4.2 shows how simple patterns map to spectra. A rectangle gives a 2-D sinc: a bright cross whose arms are longer along the rectangle's *short* side (narrow in space, wide in frequency). Shifting the rectangle leaves the spectrum identical. Rotating it rotates the spectrum. A cosine

grating is a single 2-D frequency, so its spectrum is just three dots: the DC term and a symmetric pair. A Gaussian transforms into another Gaussian.

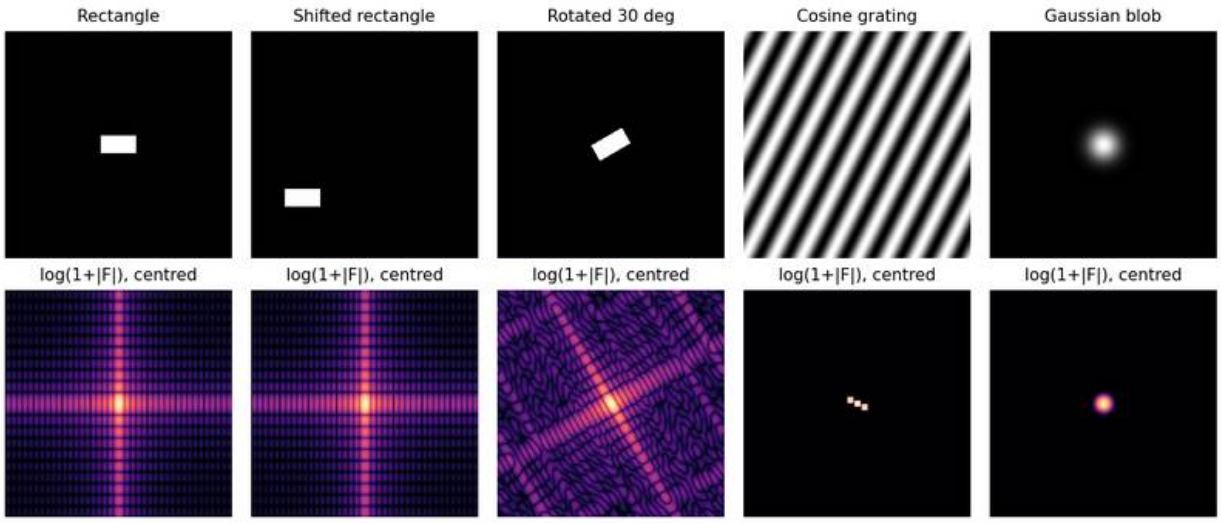


Figure 4.2 — Simple patterns (top) and their centered log spectra (bottom). Translation leaves the magnitude unchanged; rotation rotates it. The grating's three spectral dots are enlarged so they are visible.

The two-argument function `atan2(I, R)` keeps the signs of both components, so it identifies the correct quadrant; `arctan(I/R)` cannot distinguish $1 + j$ from $-1 - j$. At zero magnitude the mathematical phase is undefined, even though software returns a conventional numeric value. NumPy implements `angle` using `arctan2` [21].

The phase-versus-magnitude experiment

Which part of the transform carries the content we recognize? Take two images, keep the magnitude of one and the phase of the other, and invert. Oppenheim and Lim made this experiment famous [7].

```
import numpy as np
from skimage import color, data, img_as_float

a = img_as_float(data.camera())
b = img_as_float(color.rgb2gray(data.astronaut()))
A, B = np.fft.fft2(a), np.fft.fft2(b)
mixed = np.real(np.fft.ifft2(np.abs(A) * np.exp(1j * np.angle(B))))
# 'mixed' looks like the astronaut, not the cameraman
```

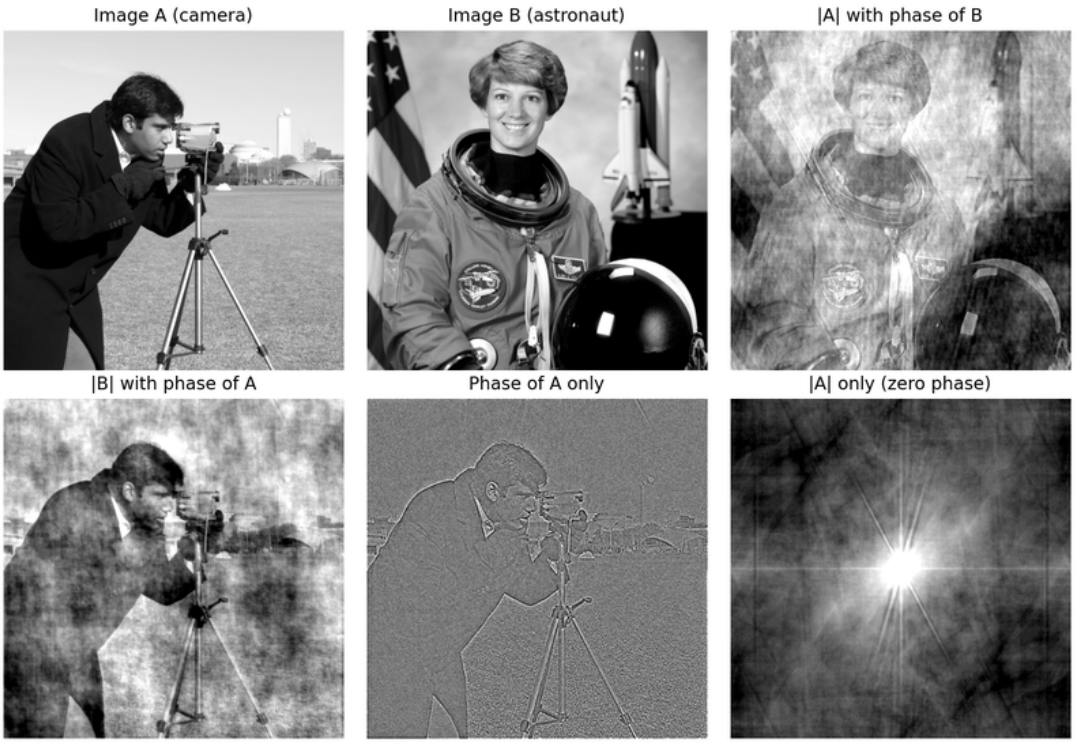


Figure 4.3 — Swapping magnitude and phase. Each hybrid looks like the image whose phase it borrowed. The phase alone (unit magnitude) keeps the edges and layout; the magnitude alone (zero phase) is an unrecognizable blob.

The result is unambiguous: **phase carries the structure**. The magnitude tells you how much energy sits at each frequency, which is roughly similar for most natural photos. The phase tells you where the waves line up, and edges are exactly the places where many waves line up. This is why the filters below are designed to leave phase untouched.

The 2-D convolution theorem

The discrete version of the convolution theorem is

$$(f \star h)(x,y) \;\Leftrightarrow\; F(u,v)\,H(u,v), \qquad (f \star h)(x,y) = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f(m,n)\,h(x-m,\;y-n)$$

where the indices $x-m$ and $y-n$ are taken **modulo** M and N. This is **circular convolution**: the DFT assumes the image repeats forever, so whatever is pushed off the right edge reappears on the left.

Wraparound error and padding

Circular convolution is not what we want; it mixes the left and right borders of the image. This is **wraparound error**. The fix is to pad both arrays with zeros so the periods are far enough apart that the copies never touch. If f is $A \times B$ and h is $C \times D$, choose a padded size $P \times Q$ with

$$P \geq A + C - 1, \qquad Q \geq B + D - 1$$

When the filter is specified in the frequency domain and is the same size as the image, the usual choice is simply $P = 2M$, $Q = 2N$.

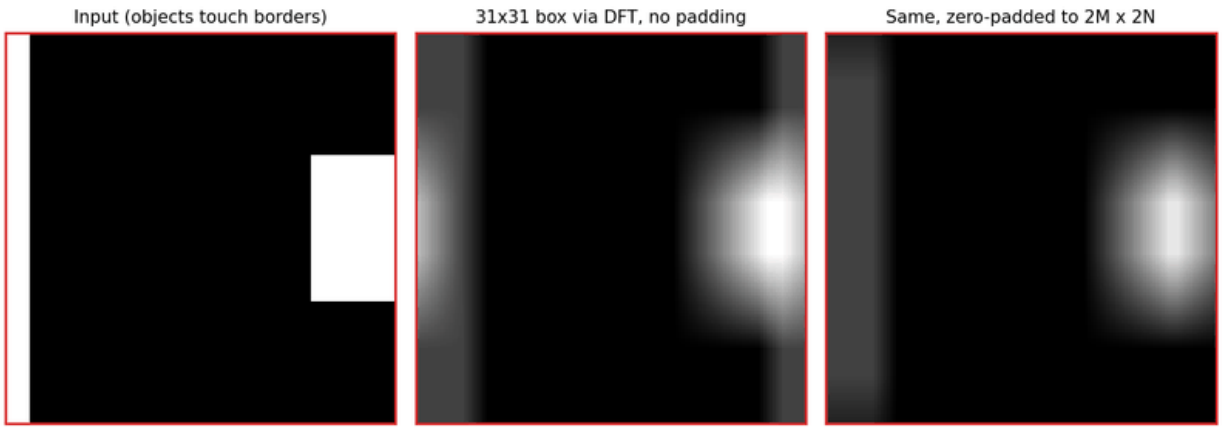


Figure 4.4 — Wraparound error. Without padding (middle), the blur leaks the right-hand block onto the left border and the left bar onto the right border. Zero-padding to twice the size (right) gives the correct linear convolution.

Basics of filtering in the frequency domain

Plain version. Transform the image, multiply each frequency by a number that says "keep" or "remove", and transform back. The list of numbers is the filter.

The filtering procedure

Given an $M \times N$ image $f(x, y)$ and a filter transfer function $H(u, v)$:

1. Choose padding sizes $P = 2M$ and $Q = 2N$.
2. Form the padded image $f_p(x, y)$ of size $P \times Q$ by appending zeros (or mirrored pixels).
3. Multiply f_p by $(-1)^{x+y}$ to center its transform.
4. Compute the DFT, $F(u, v)$.
5. Build a real, symmetric filter $H(u, v)$ of size $P \times Q$ with its center at $(P/2, Q/2)$, and form $G(u, v) = H(u, v)\, F(u, v)$.
6. Compute $g_p(x, y) = \mathrm{Re}\{\mathcal{F}^{-1}[G(u, v)]\}\,(-1)^{x+y}$.

7. Crop the top-left $M \times N$ region to get $g(x, y)$.

Taking the real part in step 6 only discards tiny imaginary values caused by floating-point rounding; with a real image and a symmetric real filter the exact result is real.

```
import numpy as np

def filter_freq(f, H_of_D):
    """Filter image f with a radially symmetric, zero-phase transfer function.
    H_of_D maps a distance array D (from the centre of the P x Q grid) to H."""
    M, N = f.shape
    P, Q = 2 * M, 2 * N                                  # 1. padding size
    fp = np.zeros((P, Q)); fp[:M, :N] = f                # 2. zero-pad
    y, x = np.mgrid[0:P, 0:Q]
    fp = fp * (-1.0) ** (x + y)                          # 3. centre the transform
    F = np.fft.fft2(fp)                                  # 4. DFT
    D = np.hypot(y - P / 2, x - Q / 2)
    G = H_of_D(D) * F                                    # 5. multiply
    gp = np.real(np.fft.ifft2(G)) * (-1.0) ** (x + y)    # 6. inverse, undo centring
    return gp[:M, :N]                                    # 7. crop
```

Zero padding darkens a thin band along the right and bottom borders after blurring, because the filter averages in black pixels. Padding by mirroring the image (`np.pad(f, ..., mode="reflect")`) avoids this, which is what the figure script for this chapter does.

Zero-phase filters

For the real, symmetric transfer functions used here, **nonnegative** gains preserve the phase wherever the gain is positive. A negative gain instead adds a phase of π; at zero gain that frequency is removed and its output phase is undefined. Thus the nonnegative lowpass and highpass masks below are zero-phase filters, while the negative Laplacian transfer function later in the chapter changes sign. Real symmetry alone does not guarantee zero phase. Designing a filter without a spatial shift is useful for keeping features aligned with the input.

Correspondence between spatial and frequency filters

By the convolution theorem, every frequency filter $H(u, v)$ has a spatial kernel $h(x, y) = \mathcal{F}^{-1}\{H(u, v)\}$, and every spatial kernel has a transfer function. The Gaussian pair is the cleanest example. In one dimension,

$$H(u) = A\, e^{-u^2/2\sigma^2} \;\Leftrightarrow\; h(x) = \sqrt{2\pi}\,\sigma A\, e^{-2\pi^2\sigma^2 x^2}$$

where σ is the width of the Gaussian in frequency. A wide frequency Gaussian corresponds to a narrow spatial kernel and vice versa. A common practical strategy is therefore to *design* in the frequency domain, where "pass below 40 cycles" is easy to state, and then *implement* a small spatial kernel that approximates h.

When is the frequency route faster? Spatial convolution with a $k \times k$ kernel costs about k^2 multiplications per pixel. The FFT route costs a fixed amount per pixel that grows only with $\log(MN)$. For small kernels (3×3, 5×5) the spatial route wins; for kernels tens of pixels wide, the FFT route wins.

Image smoothing using lowpass frequency-domain filters

Plain version. A lowpass filter keeps the slow waves and removes the fast ones, so the image gets blurrier. The way the filter drops from "keep" to "remove" matters a lot: a sudden drop creates ripples.

Let $D(u, v)$ be the distance from the center of the $P \times Q$ frequency rectangle,

$$D(u, v) = \sqrt{(u - P/2)^2 + (v - Q/2)^2}$$

and let $D_0 > 0$ be the **cutoff frequency**.

Ideal lowpass filter (ILPF)

$$H(u, v) = \begin{cases} 1 & D(u, v) \le D_0 \\ 0 & D(u, v) > D_0 \end{cases}$$

It passes everything inside a circle of radius D_0 and nothing outside. A useful way to choose D_0 is by the fraction of total power it keeps: the percentage $\alpha = 100 \sum_{D \le D_0} P(u, v) / \sum_{u,v} P(u, v)$.

Butterworth lowpass filter (BLPF)

$$H(u, v) = \frac{1}{1 + [D(u, v)/D_0]^{2n}}$$

where n is the **order**. At $D = D_0$ the filter is exactly 0.5. Small n gives a gentle roll-off; as $n \to \infty$ the BLPF approaches the ideal filter.

Gaussian lowpass filter (GLPF)

$$H(u, v) = e^{-D^2(u,v)/2D_0^2}$$

where D_0 plays the role of σ. At $D = D_0$ the filter is $e^{-1/2} \approx 0.607$.

Ringing

Figure 4.5 compares the three with the same cutoff. The ideal filter produces visible **ringing**: ghostly copies of edges repeated outward, like ripples around a stone thrown in a pond. The reason is the box-sinc pair from earlier. A sharp cutoff in frequency corresponds to a spatial kernel with a central lobe surrounded by rings of alternating sign. Convolving an edge with that kernel draws a series of faint bright and dark echoes beside it.

The Gaussian is the opposite extreme: its spatial kernel is also Gaussian, always positive, so it can never ring. The Butterworth is in between: order 1 has no ringing, order 2 has a barely visible negative lobe, and higher orders ring more and more. Order 2 is a popular compromise between a sharp cutoff and clean results.

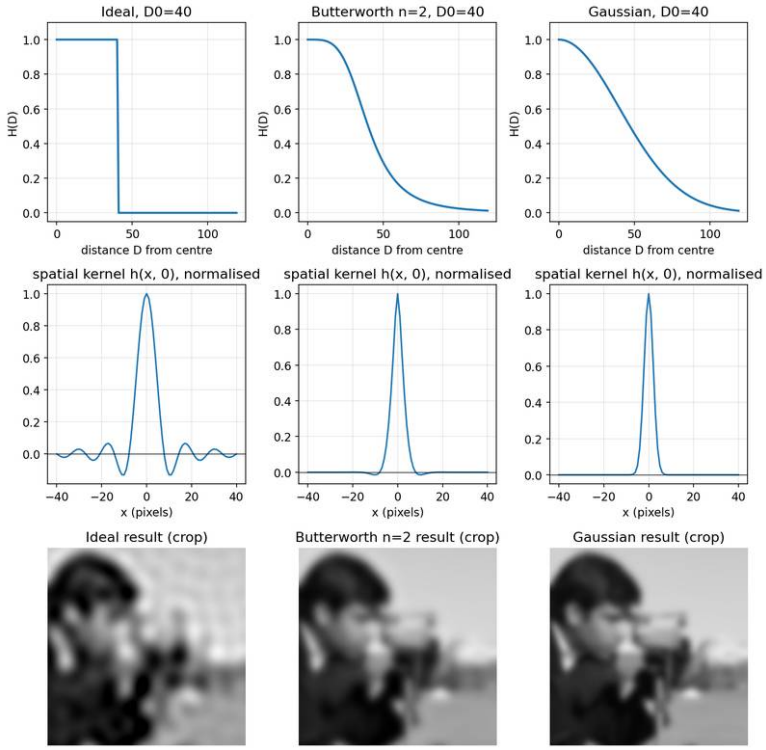


Figure 4.5 — Ideal, Butterworth ($n = 2$) and Gaussian lowpass filters with the same cutoff $D0 = 40$. Top: radial profile $H(D)$. Middle: the corresponding spatial kernel; only the ideal filter has strong negative side lobes. Bottom: the ideal filter's result shows ripples around edges (ringing); the Gaussian's does not.

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

```
D0 = 40
ideal  = lambda D: (D <= D0).astype(float)
butter = lambda D, n=2: 1 / (1 + (D / D0) ** (2 * n))
```

```
gauss  = lambda D: np.exp(-D**2 / (2 * D0**2))
smooth = filter_freq(f, gauss)
```

Lowpass filtering has many quiet uses: joining broken character strokes in a low-resolution scan, softening skin texture in portraits, and suppressing scan-line patterns in remote-sensing images.

Image sharpening using highpass filters

Plain version. A highpass filter is the opposite: it throws away the slow waves and keeps the fast ones, so only edges and texture remain. Add some of that back to the original and the image looks sharper.

Ideal, Butterworth and Gaussian highpass filters

Every highpass filter can be made from a lowpass one:

$$H_{\mathrm{HP}}(u,v) = 1 - H_{\mathrm{LP}}(u,v)$$

So the Gaussian highpass is $1 - e^{-D^2/2D_0^2}$, the Butterworth highpass is $1/(1 + [D_0/D]^{2n})$, and the ideal highpass is 0 inside the circle and 1 outside. The same ringing story applies: ideal rings, Gaussian does not. Because the highpass sets $H(0,0) = 0$, the output has zero mean, so flat regions become gray (zero) after display scaling.

The Laplacian in the frequency domain

The Laplacian $\nabla^2 f = \partial^2 f/\partial x^2 + \partial^2 f/\partial y^2$ from Chapter 3 has a transfer function. Differentiation multiplies the transform by $j2\pi u$, so differentiating twice in each direction gives

$$H(u,v) = -4\pi^2 \left[(u - P/2)^2 + (v - Q/2)^2\right] = -4\pi^2 D^2(u,v)$$

with frequencies measured in cycles per image. The sharpened image is $g(x,y) = f(x,y) - \nabla^2 f(x, y)$, with the minus sign because the center of H is negative. In practice the Laplacian output is far larger than f, so both must be scaled to comparable ranges (for example, f in $[0,1]$ and $\nabla^2 f$ divided by its maximum absolute value) before subtracting.

Unsharp masking, high-boost and high-frequency emphasis

Unsharp masking subtracts a blurred version to obtain a "mask" of details and adds it back:

$$g_{\mathrm{mask}}(x,y) = f(x,y) - f_{\mathrm{LP}}(x,y), \qquad g(x,y) = f(x,y) + k\, g_{\mathrm{mask}}(x,y)$$

where f_{LP} is a lowpass-filtered f and $k \geq 0$ is a weight ($k = 1$: unsharp masking; $k > 1$: high-boost). In the frequency domain the whole operation is one transfer function, $H = 1 + k\, H_{\mathrm{HP}}$. Generalizing gives **high-frequency emphasis**:

$$H_{\mathrm{HFE}}(u,v) = k_1 + k_2\, H_{\mathrm{HP}}(u,v)$$

where $k_1 \geq 0$ controls how much of the low frequencies (and the DC term) survives and $k_2 \geq 0$ controls how strongly high frequencies are boosted. Choosing $k_1 < 1$ also lowers overall contrast, which is why high-frequency emphasis is often followed by histogram equalization.

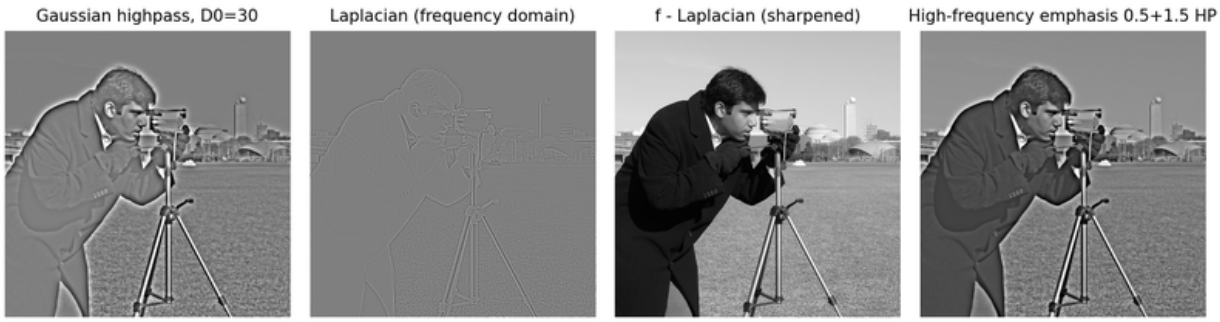


Figure 4.6 — Sharpening in the frequency domain. From left: Gaussian highpass ($D0 = 30$), the Laplacian computed with $H = -4\pi^2 D^2$, the image minus its scaled Laplacian, and high-frequency emphasis with $k1 = 0.5$, $k2 = 1.5$.

Homomorphic filtering

A simple image-formation model says each pixel is the product of **illumination** $i(x,y)$ (how much light falls on the scene) and **reflectance** $r(x,y)$ (how much the surface reflects): $f = i \cdot r$. Illumination varies slowly across a scene; reflectance changes abruptly at object boundaries. We would like to squash the illumination and boost the reflectance, but the Fourier transform of a product is not the product of transforms.

The trick, introduced by Oppenheim, Schafer and Stockham [8], is to take the logarithm first, which turns the product into a sum:

$$z(x,y) = \ln f(x,y) = \ln i(x,y) + \ln r(x,y)$$

Now filter z with a transfer function that treats low and high frequencies differently, and exponentiate:

$$s = \mathcal{F}^{-1}\{H\, Z\}, \qquad g(x,y) = e^{s(x,y)}, \qquad H(u,v) = (\gamma_H - \gamma_L)\left[1 - e^{-c\, D^2(u,v)/D_0^2}\right] + \gamma_L$$

where $\gamma_L < 1$ attenuates low frequencies (illumination), $\gamma_H > 1$ amplifies high frequencies (reflectance), c controls the steepness of the transition and D_0 its location. The result compresses the dynamic range and enhances contrast at the same time.

```
def homomorphic(f, gL=0.4, gH=1.6, c=1.0, D0=30, eps=1e-3):
    z = np.log(f + eps)                         # eps avoids log(0)
    H = lambda D: (gH - gL) * (1 - np.exp(-c * D**2 / D0**2)) + gL
    return np.exp(filter_freq(z, H)) - eps
```

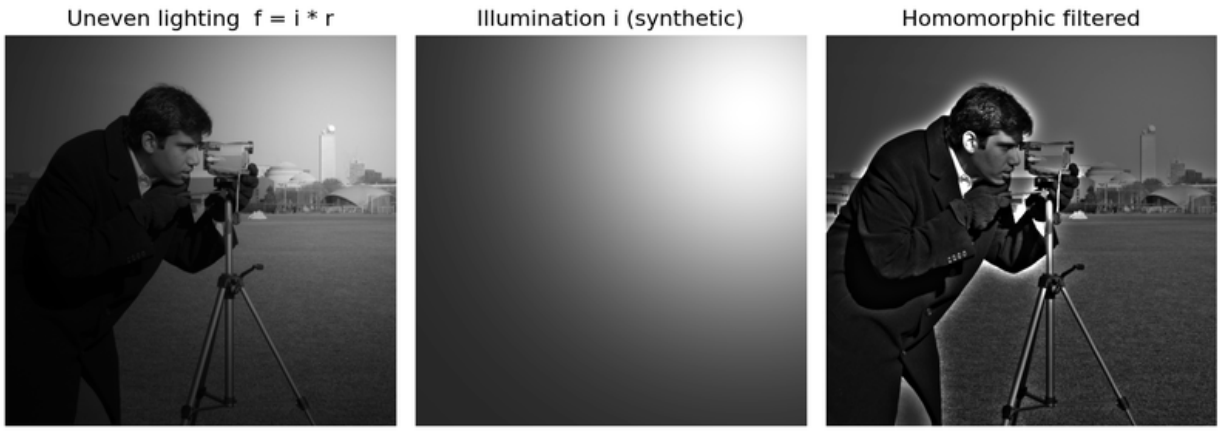


Figure 4.7 — Homomorphic filtering. A synthetic illumination field darkens the left side of the image (left, middle). Filtering in the log domain with $\gamma L = 0.4$ and $\gamma H = 1.6$ evens out the lighting and boosts detail in the dark areas (right), at the cost of mild halos around strong edges.

Selective filtering

Plain version. Sometimes the problem is not "too much detail" or "too little" but one specific, repeating pattern, like electrical interference stripes. In the spectrum such a pattern is a few bright dots. A selective filter removes just those dots and nothing else.

Bandreject and bandpass filters

A **bandreject** filter removes a ring of frequencies centered at radius C_0 with width W. The Gaussian version is

$$H_{\mathrm{BR}}(u,v) = 1 - \exp\left[-\left(\frac{D^2(u,v) - C_0^2}{D(u,v)\,W}\right)^2\right]$$

and the Butterworth version of order n is

$$H_{\mathrm{BR}}(u,v) = \frac{1}{1 + \left[\dfrac{D(u,v)\,W}{D^2(u,v) - C_0^2}\right]^{2n}}$$

where C_0 is the ring's center radius and W its width. A **bandpass** filter keeps only that ring: $H_{\mathrm{BP}} = 1 - H_{\mathrm{BR}}$. Ring-shaped filters suit noise that appears at a fixed distance from the center in all directions, which is uncommon in practice.

Notch filters

A **notch filter** acts on small neighborhoods around chosen frequencies. Because a real image's spectrum is conjugate symmetric, every notch must come with its mirror image through the center. A notch-reject filter with Q notch pairs is built as a product of highpass filters whose centers are moved to the spike locations:

$$H_{\mathrm{NR}}(u,v) = \prod_{k=1}^{Q} H_k(u,v)\, H_{-k}(u,v)$$

where H_k is a highpass filter (ideal, Butterworth or Gaussian) centered at (u_k, v_k) and H_{-k} is the same filter centered at $(-u_k, -v_k)$, both measured relative to the center of the frequency rectangle. The distances are

$$D_k(u,v) = \sqrt{(u - P/2 - u_k)^2 + (v - Q/2 - v_k)^2}, \qquad D_{-k}(u,v) = \sqrt{(u - P/2 + u_k)^2 + (v - Q/2 + v_k)^2}$$

A notch-pass filter is $1 - H_{\mathrm{NR}}$. Applying the notch-pass filter and inverting isolates the interference pattern itself, which is a useful check that you removed noise and not image content.

```
import numpy as np
from skimage import data, img_as_float

f = img_as_float(data.camera())
M, N = f.shape
y, x = np.mgrid[0:M, 0:N]
g = f + 0.2 * np.cos(2 * np.pi * (40 * x / N + 25 * y / M))   # add a ripple

G = np.fft.fftshift(np.fft.fft2(g))
u, v = y - M // 2, x - N // 2              # centred frequency coordinates
H = np.ones((M, N))
for uk, vk in [(25, 40)]:                   # spike location (row, column offset)
    for s in (+1, -1):                      # the conjugate-symmetric partner
        Dk = np.hypot(u - s * uk, v - s * vk)
        H *= 1 - np.exp(-Dk**2 / (2 * 4.0**2))   # Gaussian notch reject
out = np.real(np.fft.ifft2(np.fft.ifftshift(G * H)))
```

Padding is skipped here on purpose. The interference is periodic over the whole image, and padding would smear its spike across many frequency samples, making the notch less effective.

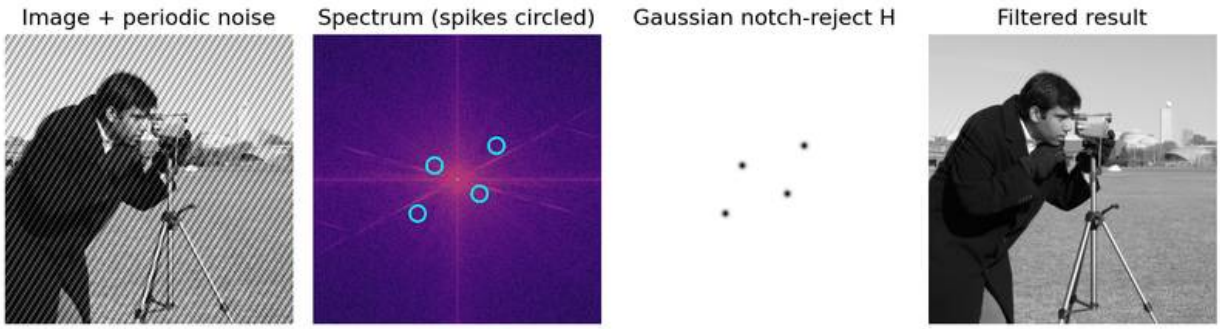


Figure 4.8 — Notch filtering of synthetic periodic noise. Two sinusoidal ripples produce four bright spikes (two conjugate pairs, circled). A Gaussian notch-reject filter with a dark dot at each spike removes them, and the interference disappears.

Drag the slider to compare the corrupted and filtered images directly:



With periodic noise



Notch filtered

Cameraman with diagonal stripe interference compared with the notch-filtered result

Implementation

Plain version. A 2-D DFT can be computed as many 1-D DFTs, an inverse DFT can be computed with the forward routine, and the FFT algorithm makes each 1-D transform dramatically faster.

Separability

The 2-D kernel $e^{-j2\pi(ux/M+vy/N)}$ factors into $e^{-j2\pi ux/M}\,e^{-j2\pi vy/N}$. So

$$F(u,v) = \sum_{x=0}^{M-1} e^{-j2\pi ux/M} \underbrace{\sum_{y=0}^{N-1} f(x,y)\, e^{-j2\pi vy/N}}_{F(x,v)}$$

where $F(x,v)$ is the 1-D DFT of row x. Compute a 1-D DFT along every row, then a 1-D DFT along every column of the result. Any 1-D FFT routine is enough to build a 2-D one.

The inverse DFT has the same form as the forward one, except for the sign of the exponent and the factor $1/MN$, so a forward FFT routine can compute the inverse as well.

Online only: Computing the inverse DFT with a forward transform — https://blog.shyandram.dev/en/learn/image-processing/frequency-domain/notes/computing-the-inverse-dft-with-a-forward-transform/

The fast Fourier transform

Computing the 1-D DFT from its definition takes about M^2 complex multiplications and additions: M outputs, each a sum of M terms. The **FFT** of Cooley and Tukey [4] reduces this to about $M \log_2 M$ for M a power of 2. The core idea is **divide and conquer**: split the sum into even- and odd-indexed samples,

$$F(u) = F_{\text{even}}(u) + W_M^u F_{\text{odd}}(u), \qquad F(u + M/2) = F_{\text{even}}(u) - W_M^u F_{\text{odd}}(u)$$

where $W_M = e^{-j2\pi/M}$, and F_{even}, F_{odd} are DFTs of length $M/2$. Each half-length DFT is split again, until the pieces have length 1. There are $\log_2 M$ levels, each costing on the order of M operations.

The ratio of the two costs, $M^2/(M \log_2 M) = M/\log_2 M$, grows quickly. For $M = 1024$ it is about 100; for a 1024×1024 image transformed directly in 2-D (cost $(MN)^2$) versus by FFT (cost $MN \log_2 MN$), the ratio is about 50,000. On a laptop the difference is easy to see:

```
import time
x = np.random.default_rng(2).standard_normal(4096)
t0 = time.perf_counter(); dft(x); t1 = time.perf_counter(); np.fft.fft(x); t2 =
 ↪  time.perf_counter()
print(f"direct {t1 - t0:.3f}s  fft {t2 - t1:.5f}s")   # e.g. ~0.7 s vs ~0.2 ms
```

In practice, use a library FFT such as FFTW [6], which handles arbitrary sizes. Sizes whose prime factors are small (2, 3, 5, 7) are fast and large primes are slower; `scipy.fft.next_fast_len` returns a good padded size.

Online only: The FFT: history and libraries — https://blog.shyandram.dev/en/learn/image-processing/frequency-domain/notes/the-fft-history-and-libraries/

Modern view

The Fourier transform is older than every other tool in this series, and it has aged unusually well. Deep learning did not replace it. Instead, Fourier ideas returned in three roles: a way to *compute* convolutions, a way to *build* layers with global reach, and a lens for *understanding* what networks learn and where they fail.

Surveys and reviews worth reading

- **Unser, "Sampling — 50 years after Shannon" [3]** reviews sampling theory from Shannon's band-limited model to spline-based formulations. Takeaway: practical systems replace the ideal sinc with compact kernels plus a matching prefilter.

- **Xu, Zhang and Luo [13]** survey the **frequency principle**: networks trained by gradient descent tend to fit low frequencies first and high frequencies later, a little like a lowpass filter that slowly opens up.
- **Kovachki et al. [15]** treat neural operators, which learn maps between functions. Parameterizing a layer by its frequency response makes it largely independent of the grid resolution.

Fourier ideas inside networks

- **Fast convolution.** By the convolution theorem, convolutions can be computed as products of FFTs; Mathieu, Henaff and LeCun [9] used this to speed up CNN training. With today's small 3×3 kernels, direct convolution usually wins, so FFT convolution is mainly used for large kernels.
- **Global filter layers.** A product in frequency touches every pixel, so a single Fourier layer has an image-wide receptive field. Global Filter Networks [11] replace self-attention with this chapter's recipe (FFT, multiply by a learned filter, inverse FFT), and the Fourier Neural Operator [14] learns filters on low frequencies, so a trained model can be evaluated on finer grids than it was trained on.
- **Spectral bias.** Deep ReLU networks learn low frequencies first; high frequencies are learned later and are less robust [12]. This is why coordinate-based networks often need their input coordinates mapped through sinusoidal Fourier features before they can represent fine texture [20].
- **Aliasing.** Strided convolutions and pooling subsample without a lowpass prefilter, so they alias, just as in Figure 4.1. Zhang [17] showed that a small blur before each downsampling step makes CNNs more stable under one-pixel shifts and, in his experiments, also improved classification accuracy; alias-free GAN generators [18] apply carefully designed lowpass filters around every resampling step.
- **Spectral fingerprints.** Upsampling in generators leaves periodic artifacts in the spectrum, much like the spikes in Figure 4.8; Frank et al. [19] used them to build a simple and accurate detector of GAN-generated images.

Online only: Fourier ideas in deep learning in more detail — https://blog.shyandram.dev/en/learn/image-processing/frequency-domain/papers/fourier-ideas-in-deep-learning-in-more-detail/

What did not change

Deep learning has not replaced classical frequency filtering for well-understood problems. Removing known periodic interference, anti-aliasing before resizing, fast large-kernel blurs, and homomorphic illumination correction are still done with the explicit filters in this chapter, because they are exact, fast, and need no training data. What changed is that these same ideas are now built into network architectures and used to explain their behavior.

Key takeaways

- The Fourier transform rewrites a signal as a sum of waves. Low frequencies carry smooth structure; high frequencies carry edges, texture and noise.
- Sampling copies the spectrum periodically. Sample faster than twice the highest frequency (the Nyquist rate), or blur first, otherwise aliasing and moiré appear and cannot be undone.
- The DFT treats the image as periodic. That gives circular convolution, so pad to at least $A+C-1$ (typically $2M \times 2N$) to avoid wraparound error.
- Translation changes only the phase; rotation rotates the spectrum; real images have conjugate-symmetric spectra; and the phase carries most of the recognizable structure.
- Filtering is "transform, multiply by a real symmetric H, invert, crop". Sharp cutoffs cause ringing; Gaussian filters do not; Butterworth sits in between.
- Highpass, Laplacian, high-frequency emphasis and homomorphic filters sharpen or rebalance lighting; notch filters remove periodic interference precisely.
- The FFT lowers the cost from $O(M^2)$ to $O(M \log M)$ per 1-D transform, and separability builds 2-D transforms from 1-D ones.
- In deep learning, Fourier ideas power fast convolution, global filter layers and neural operators, and explain spectral bias, aliasing in CNNs and GAN artifacts.

Exercises

1. A 1-D signal contains components at 3, 7 and 12 cycles per meter. You sample it every 0.1 m. Which components survive correctly, and at what frequency does each aliased component appear?

   Hint

   The sampling rate is 10 samples per meter, so the Nyquist limit is 5 cycles per meter. The 3 cycle component is fine. Fold the others back into $[-5, 5]$ by subtracting multiples of 10: 7 appears as $7-10 = -3$, i.e. 3 cycles per meter, and 12 appears as $12 - 10 = 2$ cycles per meter. The 7 cycle component is now indistinguishable from the real 3 cycle one.

2. Show that if $f(x, y)$ is real and symmetric, $f(x, y) = f(-x, -y)$ (indices modulo M, N), then its DFT is real. Verify numerically with a symmetric array in NumPy.

   Hint

   Conjugate symmetry gives $F(-u, -v) = F^*(u, v)$ for any real f. Symmetry of f gives $F(-u, -v) = F(u, v)$ by substituting $x \to -x, y \to -y$ in the definition. Together, $F = F^*$, so F is real. For the check, build `g = a + np.roll(a[::-1, ::-1], 1, axis=(0, 1))` for a random `a` and confirm `np.abs(np.fft.fft2(g).imag).max()` is near zero.

3. Using the `filter_freq` function, filter the camera image with a Butterworth lowpass filter of orders 1, 2, 5 and 20 at $D_0 = 30$. At what order does ringing first become clearly visible? Plot the central row of each filter's spatial kernel and relate it to what you see.

Hint

Obtain the kernel by inverse-transforming the centered filter and shifting it back to the center. Ringing appears when the kernel develops noticeable negative side lobes. Order 1 has none, order 2 has a tiny one, and by order 5 to 20 the lobes approach those of the ideal filter.

4. You filter a 300×400 image with a 51×51 spatial kernel using FFTs. What is the minimum padded size that avoids wraparound error? Would you choose that size or a slightly larger one, and why?

Hint

The minimum is $(300 + 51 - 1) \times (400 + 51 - 1) = 350 \times 450$. Both are fine, but sizes with only small prime factors are faster for FFT libraries. $350 = 2 \cdot 5^2 \cdot 7$ and $450 = 2 \cdot 3^2 \cdot 5^2$ already are, so they are good choices; `scipy.fft.next_fast_len` confirms this.

5. Photograph (or synthesize) an image with two different periodic interference patterns. Design a notch-reject filter that removes only one of them, then use the corresponding notch-pass filter to display the pattern you removed. How do you choose the notch radius?

Hint

Locate the spikes in the log spectrum, ignoring the DC peak and the axes. Remember each spike's mirror twin. A radius just large enough to cover the spike's spread works best: too small leaves residual stripes, too large removes image content near those frequencies. The notch-pass output should look like clean stripes with no trace of the scene.

6. Explain, using the ideas in this chapter, why inserting a blur before every stride-2 layer can make a CNN's output more stable when the input image is shifted by one pixel.

Hint

Stride-2 subsampling halves the sampling rate, so frequencies above the new Nyquist limit fold back as aliases. A one-pixel shift changes the phase of those components, and after aliasing that changes the low-frequency output, so the next layers see a different signal. A lowpass filter removes those frequencies first, so a shift changes the output much less. See [17].

References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018, Ch. 4. publisher page
2. C. E. Shannon, "Communication in the presence of noise," *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, 1949. doi
3. M. Unser, "Sampling—50 years after Shannon," *Proceedings of the IEEE*, vol. 88, no. 4, pp. 569–587, 2000. doi

4. J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex Fourier series,” *Mathematics of Computation*, vol. 19, no. 90, 1965. doi
5. M. T. Heideman, D. H. Johnson and C. S. Burrus, “Gauss and the history of the fast Fourier transform,” *IEEE ASSP Magazine*, vol. 1, no. 4, pp. 14–21, 1984. doi
6. M. Frigo and S. G. Johnson, “The design and implementation of FFTW3,” *Proceedings of the IEEE*, vol. 93, no. 2, 2005. doi
7. A. V. Oppenheim and J. S. Lim, “The importance of phase in signals,” *Proceedings of the IEEE*, vol. 69, no. 5, 1981. doi
8. A. V. Oppenheim, R. W. Schafer and T. G. Stockham, “Nonlinear filtering of multiplied and convolved signals,” *Proceedings of the IEEE*, vol. 56, no. 8, pp. 1264–1291, 1968. doi
9. M. Mathieu, M. Henaff and Y. LeCun, “Fast training of convolutional networks through FFTs,” arXiv:1312.5851, 2013. arXiv
10. L. Chi, B. Jiang and Y. Mu, “Fast Fourier convolution,” *Advances in Neural Information Processing Systems* (*NeurIPS*), vol. 33, 2020. proceedings
11. Y. Rao, W. Zhao, Z. Zhu, J. Lu and J. Zhou, “Global filter networks for image classification,” *NeurIPS*, 2021. arXiv
12. N. Rahaman, A. Baratin, D. Arpit, F. Draxler, M. Lin, F. A. Hamprecht, Y. Bengio and A. Courville, “On the spectral bias of neural networks,” *ICML*, 2019. arXiv
13. Z.-Q. J. Xu, Y. Zhang and T. Luo, “Overview frequency principle/spectral bias in deep learning,” arXiv:2201.07395, 2022. arXiv
14. Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart and A. Anandkumar, “Fourier neural operator for parametric partial differential equations,” arXiv:2010.08895, 2020. arXiv
15. N. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. Stuart and A. Anandkumar, “Neural operator: Learning maps between function spaces,” *Journal of Machine Learning Research*, vol. 24, 2023. arXiv
16. K. Xu, M. Qin, F. Sun, Y. Wang, Y.-K. Chen and F. Ren, “Learning in the frequency domain,” *CVPR*, 2020. arXiv
17. R. Zhang, “Making convolutional networks shift-invariant again,” *ICML*, 2019. arXiv
18. T. Karras, M. Aittala, S. Laine, E. Härkönen, J. Hellsten, J. Lehtinen and T. Aila, “Alias-free generative adversarial networks,” arXiv:2106.12423, 2021. arXiv
19. J. Frank, T. Eisenhofer, L. Schönherr, A. Fischer, D. Kolossa and T. Holz, “Leveraging frequency analysis for deep fake image recognition,” *ICML*, 2020. arXiv
20. M. Tancik, P. P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. T. Barron and R. Ng, “Fourier features let networks learn high frequency functions in low dimensional domains,” arXiv:2006.10739, 2020. arXiv
21. NumPy developers, “numpy.angle,” NumPy reference documentation, accessed 5 November 2026. API documentation.

IP 05 • Image Restoration and Reconstruction

Model how an image was damaged by blur and noise, then undo it with order-statistic, notch, inverse, Wiener and constrained least squares filters, and rebuild CT slices from projections.

Online edition: https://blog.shyandram.dev/en/learn/image-processing/restoration-reconstruction/

What you'll learn

- How to write image damage as a model, $g = h \star f + \eta$, and why that model separates *restoration* from *enhancement*
- The common noise models, how to recognise them from a histogram, and how to measure their parameters from a flat patch
- Which spatial filter to use for which noise: mean filters, order-statistic filters, and adaptive filters
- How to remove periodic interference with notch filters in the frequency domain
- How to estimate a blur and undo it with inverse, Wiener, constrained least squares and geometric mean filters, and why the naive inverse fails
- How a CT scanner turns a set of 1-D projections into a 2-D slice: the Radon transform, the Fourier-slice theorem and filtered backprojection

The big picture

Chapter 3 and Chapter 4 improved images so they *look* better. This chapter asks a stricter question: *what did the image look like before it was damaged?* To answer it we need a model of the damage. Once we have the model, undoing it becomes a well-posed (or at least a well-understood) mathematical problem. The chapter follows the arrangement of Gonzalez and Woods [1].

> **Explain like I'm five**
>
> Imagine someone smudged your drawing with a wet finger and then sprinkled some sand on it. If you just "make it prettier", you might colour over everything. But if you know *how* the finger moved and *what* the sand looks like, you can carefully brush off the sand and un-smudge the lines. Restoration is the careful version: first find out what happened, then reverse exactly that.

Restoration matters wherever the true scene is what counts: astronomy, microscopy, forensic photos, medical imaging and every phone camera pipeline. The last part of the chapter, reconstruction from projections, is the same idea pushed to the extreme. A CT scanner never sees the slice at all. It sees only shadows of it from many angles, and we must *reconstruct* the slice from them.

A model of the image degradation/restoration process

In plain words: the camera sees a slightly blurred copy of the scene, and then random noise is added on top. Restoration tries to get back the scene.

We write the degradation as

$$g(x,y) = h(x,y) \star f(x,y) + \eta(x,y),$$

where $f(x,y)$ is the ideal (unknown) image, $h(x,y)$ is the **degradation function**, also called the point spread function (PSF), $\star$ is 2-D convolution, $\eta(x,y)$ is additive noise, and $g(x,y)$ is the image we actually observe. By the convolution theorem from Chapter 4, the same model in the frequency domain is

$$G(u,v) = H(u,v)\,F(u,v) + N(u,v),$$

where capital letters denote the 2-D discrete Fourier transforms (DFTs) of the matching lowercase functions, and (u,v) are frequency coordinates. $H(u,v)$ is called the **optical transfer function** (OTF).

The goal of restoration is an estimate $\hat{f}(x,y)$ that is as close as possible to $f(x,y)$. The more we know about h and η, the closer we can get.

Restoration versus enhancement

Enhancement (Chapter 3 and 4) is largely *subjective*: we change contrast or sharpen until the result looks good to a viewer. Restoration is *objective*: it uses a model of the degradation and judges the result by how close it gets to the true image, usually with a measurable criterion such as mean squared error. Some tools appear in both chapters (a median filter, for instance), but the reason for using them differs. In restoration we pick the filter *because* the noise model tells us it is the right one.

This chapter first handles the case $h = \delta$ (no blur, only noise), then adds blur.

Noise models

In plain words: noise is the random "grain" on top of the picture. Different sensors and situations produce different kinds of grain, and each kind has its own fingerprint in the histogram.

Spatial and frequency properties of noise

Two questions describe noise. *Spatially*, is the noise at one pixel related to the noise at its neighbours, and is it related to the image itself? In this chapter we assume, unless stated otherwise, that noise is independent of position and uncorrelated with the image. *In frequency*, how is the noise power spread across (u,v)? Noise whose Fourier spectrum is roughly constant is called

white noise, by analogy with white light containing all visible frequencies. Periodic noise is the opposite: its power sits at a few isolated frequencies.

Under these assumptions the noise is fully described by the probability density function (PDF) of its values, written $p(z)$, where z is an intensity.

The common PDFs

Gaussian. The most widely used model, because it is mathematically convenient and fits electronic sensor noise well:

$$p(z) = \frac{1}{\sqrt{2\pi}\,\sigma}\, e^{-(z-\bar{z})^2/2\sigma^2},$$

where $\bar{z}$ is the mean and σ the standard deviation (so σ^2 is the variance). About 68% of values fall within $\bar{z} \pm \sigma$ and about 95% within $\bar{z} \pm 2\sigma$.

Rayleigh. A one-sided, skewed density that describes noise in range imaging:

$$p(z) = \begin{cases} \dfrac{2}{b}(z-a)\, e^{-(z-a)^2/b}, & z \geq a \\ 0, & z < a \end{cases} \qquad \bar{z} = a + \sqrt{\pi b/4}, \quad \sigma^2 = \frac{b(4-\pi)}{4},$$

where a shifts the starting point and $b > 0$ controls the spread.

Erlang (gamma). Also skewed, with a longer tail:

$$p(z) = \begin{cases} \dfrac{a^b z^{b-1}}{(b-1)!}\, e^{-az}, & z \geq 0 \\ 0, & z < 0 \end{cases} \qquad \bar{z} = \frac{b}{a}, \quad \sigma^2 = \frac{b}{a^2},$$

where $a > 0$ is a rate and b is a positive integer (the shape). Allowing non-integer b gives the general gamma density.

Exponential. The Erlang density with $b = 1$:

$$p(z) = a\, e^{-az}\ (z \geq 0), \qquad \bar{z} = \frac{1}{a}, \quad \sigma^2 = \frac{1}{a^2}.$$

It appears in laser imaging (speckle intensity).

Uniform. Every value in an interval $[a, b]$ is equally likely:

$$p(z) = \frac{1}{b-a}\ (a \leq z \leq b), \qquad \bar{z} = \frac{a+b}{2}, \quad \sigma^2 = \frac{(b-a)^2}{12}.$$

It is mainly useful for simulations (quantization error is approximately uniform).

Salt-and-pepper (impulse) noise. A pixel is either untouched, or replaced by a very bright value (salt) with probability P_s, or by a very dark value (pepper) with probability P_p. The total fraction of corrupted pixels is $P = P_s + P_p$, called the **noise density**. Faulty sensor elements and transmission bit errors produce this kind of noise. Unlike the others, it is not additive: it *replaces* pixel values.

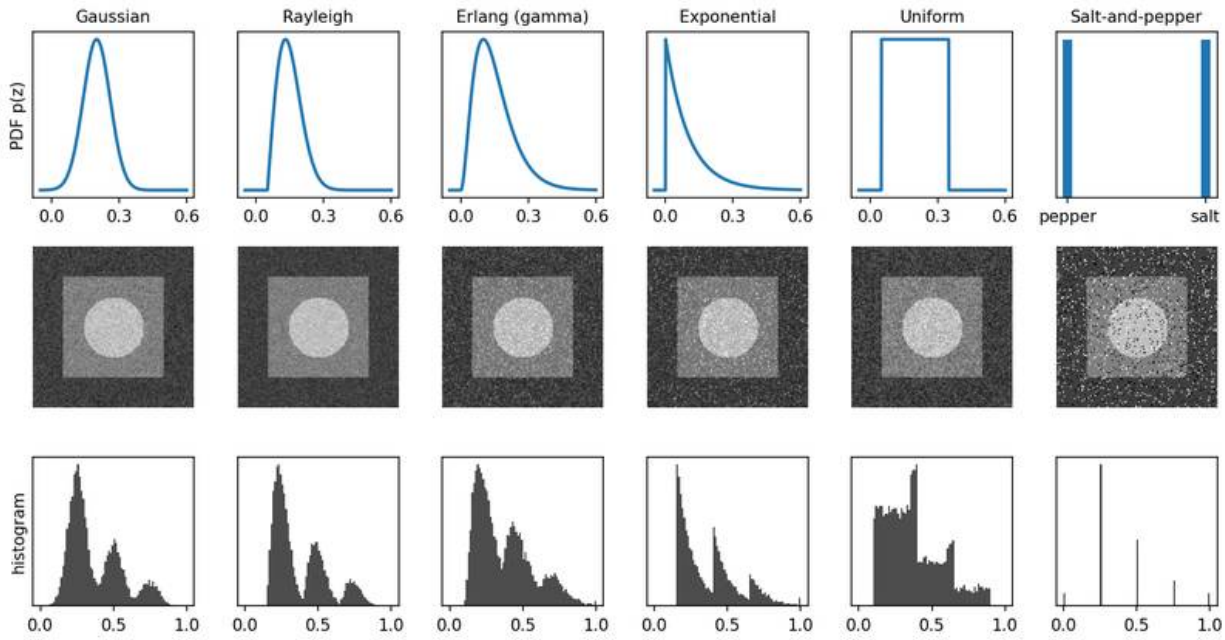


Figure 5.1 — Top: the six noise PDFs. Middle: a test pattern with three flat gray levels corrupted by each noise. Bottom: the histograms. The noisy images look similar, but each histogram has three copies of the noise PDF, one per gray level, so the histogram identifies the noise type. Impulse noise is the exception: it adds spikes at 0 and 1.

Periodic noise

Electrical or electromechanical interference during image capture can add a sinusoid such as $A\sin(2\pi(u_0 x/M + v_0 y/N))$ to the image, where A is its amplitude, (u_0, v_0) its frequency, and $M \times N$ the image size. It is *spatially dependent*, so a PDF does not describe it. In the Fourier spectrum, however, each sinusoid is a pair of bright spikes at $\pm(u_0, v_0)$, which makes it easy to remove there (see the section on periodic noise reduction below).

Estimating noise parameters

If you can control the imaging system, photograph a flat, uniformly lit target: everything you see is noise. If you only have the images, find a small region S that should be constant (sky, a blank wall, the background) and compute its histogram $p_S(z_i)$, the fraction of pixels in S with intensity z_i. Then

$$\bar{z} = \sum_i z_i \, p_S(z_i), \qquad \sigma^2 = \sum_i (z_i - \bar{z})^2 \, p_S(z_i).$$

The *shape* of the patch histogram tells you which PDF to use. The mean and variance then give the parameters (a and b for Rayleigh, Erlang, and so on). For impulse noise, count the pixels at the extreme values to estimate P_s and P_p.

```
import numpy as np
from skimage import data, img_as_float

rng = np.random.default_rng(0)
f = img_as_float(data.camera())                  # values in [0, 1]

g_gauss = f + rng.normal(0.0, 0.05, f.shape)     # additive Gaussian, sigma = 0.05
g_unif = f + rng.uniform(-0.1, 0.1, f.shape)     # additive uniform on [-0.1, 0.1]
g_sp = f.copy()
r = rng.random(f.shape)
g_sp[r < 0.05] = 0.0                             # 5 % pepper
g_sp[r > 0.95] = 1.0                             # 5 % salt

# Estimate the noise from a patch that should be flat (here: sky, top-left)
patch = g_gauss[10:60, 10:60]
print(f"mean = {patch.mean():.3f}, std = {patch.std():.3f}")   # std close to 0.05
```

Restoration in the presence of noise only — spatial filtering

In plain words: when the only problem is noise, replace every pixel by a smart summary of its neighbourhood. The best summary depends on the kind of noise.

With no blur, the model reduces to $g(x,y) = f(x,y) + \eta(x,y)$ (or, for impulse noise, a replacement). Below, S_{xy} is an $m \times n$ window centred at (x,y), and (r,c) runs over the pixels in it.

Mean filters

Arithmetic mean — the plain average:

$$\hat{f}(x,y) = \frac{1}{mn} \sum_{(r,c)\in S_{xy}} g(r,c).$$

It reduces Gaussian or uniform noise but blurs edges.

Geometric mean — the mn-th root of the product:

$$\hat{f}(x,y) = \Big[\prod_{(r,c)\in S_{xy}} g(r,c) \Big]^{1/mn}.$$

Smoothing is similar to the arithmetic mean, but less detail is lost.

Harmonic mean:

$$\hat{f}(x,y) = \frac{mn}{\sum_{(r,c)\in S_{xy}} 1/g(r,c)}.$$

It handles salt noise and Gaussian-like noise well but fails badly on pepper noise, because a single near-zero pixel dominates the sum of reciprocals.

Contraharmonic mean of order Q:

$$\hat{f}(x,y) = \frac{\sum_{(r,c)\in S_{xy}} g(r,c)^{Q+1}}{\sum_{(r,c)\in S_{xy}} g(r,c)^{Q}}.$$

Here Q is the order. Positive Q removes pepper noise, negative Q removes salt noise, and it cannot remove both at once. $Q = 0$ gives the arithmetic mean and $Q = -1$ the harmonic mean. Using the wrong sign is disastrous, as Figure 5.2 shows.

Order-statistic filters

These filters sort the pixels in S_{xy} and pick from the sorted list.

- **Median:** $\hat{f}(x,y) = \text{median}_{(r,c)\in S_{xy}}\, g(r,c)$. The best general tool for impulse noise. It removes isolated outliers while blurring much less than a mean filter of the same size.
- **Max** and **min:** the largest and smallest value. The max filter removes pepper (dark) noise, the min filter removes salt (bright) noise.
- **Midpoint:** $\frac{1}{2}(\max + \min)$. It combines ordering with averaging and suits Gaussian or uniform noise.
- **Alpha-trimmed mean:** drop the $d/2$ lowest and $d/2$ highest values, average the remaining $mn - d$ values $g_R(r,c)$:

$$\hat{f}(x,y) = \frac{1}{mn-d} \sum_{(r,c)\in S_{xy}} g_R(r,c).$$

With $d = 0$ it is the arithmetic mean; with $d = mn - 1$ it is the median. In between, it handles *mixtures*, such as Gaussian noise plus some impulses.

```
import numpy as np
from scipy import ndimage as ndi

def arithmetic_mean(g, m=3):
    return ndi.uniform_filter(g, m)

def geometric_mean(g, m=3):
    return np.exp(ndi.uniform_filter(np.log(g + 1e-6), m))

def harmonic_mean(g, m=3):
    return 1.0 / ndi.uniform_filter(1.0 / (g + 1e-6), m)

def contraharmonic_mean(g, m=3, Q=1.5):
    g = g + 1e-6                                  # avoid 0 ** negative
    return ndi.uniform_filter(g ** (Q + 1), m) / ndi.uniform_filter(g ** Q, m)

def alpha_trimmed_mean(g, m=5, d=6):
    def trim(v):
        v = np.sort(v)
```

```
        return v[d // 2: v.size - d // 2].mean()
    return ndi.generic_filter(g, trim, size=m)

median = lambda g, m=3: ndi.median_filter(g, m)
maxf = lambda g, m=3: ndi.maximum_filter(g, m)    # removes pepper
minf = lambda g, m=3: ndi.minimum_filter(g, m)    # removes salt
midpoint = lambda g, m=3: 0.5 * (maxf(g, m) + minf(g, m))
```

Adaptive filters

The filters above treat every pixel the same way. Adaptive filters change their behaviour according to local statistics in S_{xy}, and usually perform better at a modest extra cost.

Adaptive, local noise reduction filter. Let σ_η^2 be the noise variance (estimated once, for the whole image), and let $\bar{z}_S$ and σ_S^2 be the local mean and local variance in S_{xy}. The filter is

$$\hat{f}(x,y) = g(x,y) - \frac{\sigma_\eta^2}{\sigma_S^2}\left[g(x,y) - \bar{z}_S\right].$$

Read it as three cases. If $\sigma_\eta^2 = 0$ there is no noise, and the filter returns g unchanged. Where the local variance is much larger than the noise variance, the window probably contains an edge, so the filter stays close to g and keeps the edge. Where the two variances are equal, the window is a flat area, so the filter returns the local mean. In practice the ratio is clipped to at most 1, because a local variance estimate can come out smaller than σ_η^2 by chance.

Adaptive median filter. A fixed median filter works well while impulses are sparse; at high noise density a window may hold more impulses than real pixels, and a bigger fixed window blurs everything. The adaptive median filter grows the window when needed, and it leaves uncorrupted pixels alone. Let $z_{\min}$, $z_{\max}$, z_{med} be the minimum, maximum and median in S_{xy}, z_{xy} the centre pixel, and $S_{\max}$ the largest allowed window size [2].

- **Level A:** if $z_{\min} < z_{\text{med}} < z_{\max}$, the median is not an impulse, so go to Level B. Otherwise enlarge the window; if the size exceeds $S_{\max}$, output z_{med}; else repeat Level A.
- **Level B:** if $z_{\min} < z_{xy} < z_{\max}$, the centre pixel is not an impulse, so output z_{xy} unchanged. Otherwise output z_{med}.

Three goals are met at once: remove impulses, smooth other noise, and reduce the thinning and blurring of edges caused by a plain median.

```
import numpy as np
from scipy import ndimage as ndi

def adaptive_local(g, noise_var, m=7):
    """Adaptive local noise reduction: f = g - (s_n^2 / s_L^2)(g - m_L)."""
    mean = ndi.uniform_filter(g, m)
    var = ndi.uniform_filter(g * g, m) - mean ** 2
    ratio = np.minimum(noise_var / np.maximum(var, 1e-12), 1.0)  # never above 1
```

```
    return g - ratio * (g - mean)

def adaptive_median(g, s_max=7):
    out, done = g.copy(), np.zeros(g.shape, bool)
    for s in range(3, s_max + 1, 2):
        zmin, zmax = ndi.minimum_filter(g, s), ndi.maximum_filter(g, s)
        zmed = ndi.median_filter(g, s)
        stage_a = (zmin < zmed) & (zmed < zmax) & ~done          # median is not an impulse
        keep = (zmin < g) & (g < zmax)                            # centre pixel is not an
         ↪  impulse
        out[stage_a] = np.where(keep, g, zmed)[stage_a]
        done |= stage_a
    out[~done] = zmed[~done]                                      # window grew to s_max
    return out
```

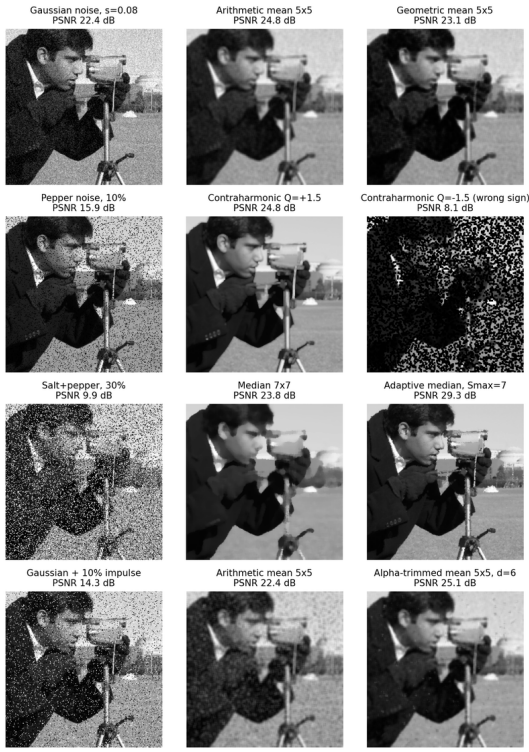


Figure 5.2 — Matching the filter to the noise. Row 1: Gaussian noise, arithmetic and geometric mean. Row 2: pepper noise; the contraharmonic filter with $Q = +1.5$ removes it, while $Q = -1.5$ makes it far worse. Row 3: 30% salt-and-pepper noise; the 7×7 median blurs and smears detail, the adaptive median keeps much more of it. Row 4: Gaussian plus impulse noise; the alpha-trimmed mean beats the plain mean. PSNR is peak signal-to-noise ratio against the clean image (higher is better).

Periodic noise reduction using frequency-domain filtering

In plain words: periodic noise is a few pure "tones" hidden in the image. In the spectrum they show up as bright dots, so we cut out exactly those dots and leave everything else.

Notch reject and notch pass filters

A **notch filter** rejects (or passes) frequencies in small neighbourhoods around chosen centres. Because the DFT of a real image is conjugate-symmetric, notches always come in pairs, at (u_k, v_k) and $(-u_k, -v_k)$ relative to the centre of the spectrum. A notch reject filter with Q notch pairs is a product of highpass filters whose centres have been moved to the notches:

$$H_{NR}(u,v) = \prod_{k=1}^{Q} H_k(u,v)\, H_{-k}(u,v),$$

where H_k and H_{-k} are highpass filters (ideal, Butterworth or Gaussian, as in Chapter 4) centred at (u_k, v_k) and $(-u_k, -v_k)$. For example, a Gaussian notch uses $H_k = 1 - e^{-D_k^2/2D_0^2}$, with $D_k(u,v)$ the distance from (u,v) to (u_k, v_k) and D_0 the notch radius. The matching notch pass filter is

$$H_{NP}(u,v) = 1 - H_{NR}(u,v).$$

The notch pass filter is useful on its own: applying it to G and transforming back gives an image of the interference pattern alone. Notch filters of other shapes (for instance a thin line along an axis) remove scan-line patterns.

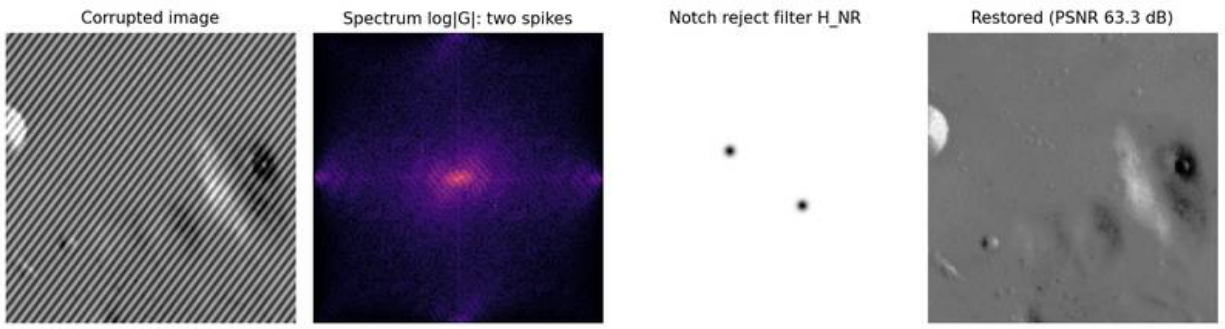


Figure 5.3 — A sinusoidal interference pattern (left) is two isolated spikes in the spectrum. Two Gaussian notches remove them, and the inverse DFT returns the clean image. The restoration here is almost perfect because the interference is a single exact sinusoid; real interference usually spreads over several frequencies.

```
import numpy as np
from skimage import data, img_as_float

f = img_as_float(data.moon())
M, N = f.shape
y, x = np.mgrid[:M, :N]
g = f + 0.2 * np.sin(2 * np.pi * (40 * y / M + 25 * x / N))    # periodic interference
```

```
G = np.fft.fftshift(np.fft.fft2(g))
U, V = np.mgrid[-M // 2:M // 2, -N // 2:N // 2]
H = np.ones((M, N))
for u0, v0 in [(40, 25), (-40, -25)]:                          # a spike and its mirror
    D2 = (U - u0) ** 2 + (V - v0) ** 2
    H *= 1 - np.exp(-D2 / (2 * 3.0 ** 2))                      # Gaussian notch, radius ~3
f_hat = np.real(np.fft.ifft2(np.fft.ifftshift(H * G)))
```

Online only: Optimum notch filtering — https://blog.shyandram.dev/en/learn/image-processing/restoration-reconstruction/notes/optimum-notch-filtering/

Linear, position-invariant degradations

In plain words: we assume the blur is "fair". Twice as much light gives twice as much blurred light, and the blur is the same in the top-left corner as in the bottom-right.

Write the degradation (without noise) as an operator $\mathcal{H}$, so $g = \mathcal{H}[f] + \eta$. $\mathcal{H}$ is **linear** if

$$\mathcal{H}[af_1 + bf_2] = a\,\mathcal{H}[f_1] + b\,\mathcal{H}[f_2]$$

for any images f_1, f_2 and constants a, b. It is **position invariant** if shifting the input only shifts the output:

$$\mathcal{H}[f(x-\alpha, y-\beta)] = g(x-\alpha, y-\beta)$$

for any shift (α, β). For such an operator, write f as a sum of shifted impulses. Linearity lets us apply $\mathcal{H}$ to each impulse separately, and position invariance says every impulse produces the same response h, just shifted. Summing these responses is exactly convolution:

$$g(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(\alpha,\beta)\,h(x-\alpha, y-\beta)\,d\alpha\,d\beta + \eta(x,y) = (h \star f)(x,y) + \eta(x,y).$$

Here $h(x,y) = \mathcal{H}[\delta(x,y)]$ is the **impulse response**, which in optics is the point spread function: the image of a single point of light. This is why restoring a blurred image is often called **deconvolution**, and why the filters below are built in the frequency domain.

Many real degradations are only approximately LPI. A lens may blur more at the image corners, and an object moving in front of a still background blurs differently from the background. Space-variant restoration exists, but it is far more expensive; the common workaround is to split the image into tiles that are roughly LPI.

Estimating the degradation function

In plain words: before we can undo the blur, we must know the blur. We can look for clues in the picture, test the camera ourselves, or derive the blur from physics.

By observation

Find a small region of the blurred image $g_s(x, y)$ with simple, strong structure, such as an edge or a small bright object, where signal is much stronger than noise. Build by hand (or by sharpening and judgment) an estimate $\hat{f}_s(x, y)$ of what that region should look like. Then

$$H_s(u, v) = \frac{G_s(u, v)}{\hat{F}_s(u, v)}$$

estimates the OTF from that small region. If the blur is position invariant, H_s tells us the shape of H for the whole image, scaled up to full size. This is laborious and used mainly when the image is all there is, for example a historical photograph.

By experimentation

If you have the same (or an equivalent) imaging system, image a tiny bright dot under the same settings. A small enough dot approximates an impulse of strength A, whose Fourier transform is the constant A. The observed image of the dot then gives the OTF directly:

$$H(u, v) = \frac{G(u, v)}{A}.$$

Microscopists do exactly this with sub-resolution fluorescent beads.

By mathematical modeling

Sometimes physics gives the answer. Two classic models are used throughout this chapter.

Atmospheric turbulence. Long-exposure imaging through turbulent air is often modelled by

$$H(u, v) = e^{-k\,(u^2+v^2)^{5/6}},$$

where k is a constant that grows with the strength of the turbulence. Apart from the $5/6$ power it looks like a Gaussian lowpass filter, and it never reaches exactly zero.

Uniform linear motion blur. Suppose the image moves at constant speed during an exposure of length T, travelling a total distance a in x and b in y. The sensor adds up every shifted copy of the scene, and the transfer function of that averaging is

$$H(u, v) = \frac{T}{\pi(ua + vb)} \sin\big[\pi(ua + vb)\big]\, e^{-j\pi(ua+vb)}.$$

Online only: Deriving the motion-blur transfer function — https://blog.shyandram.dev/en/learn/image-processing/restoration-reconstruction/notes/deriving-the-motion-blur-transfer-function/

This is a sinc function along the direction of motion, with a linear phase term. Crucially, it is exactly **zero** wherever $ua + vb$ is a non-zero integer, so whole lines of frequencies are destroyed (Figure 5.4).

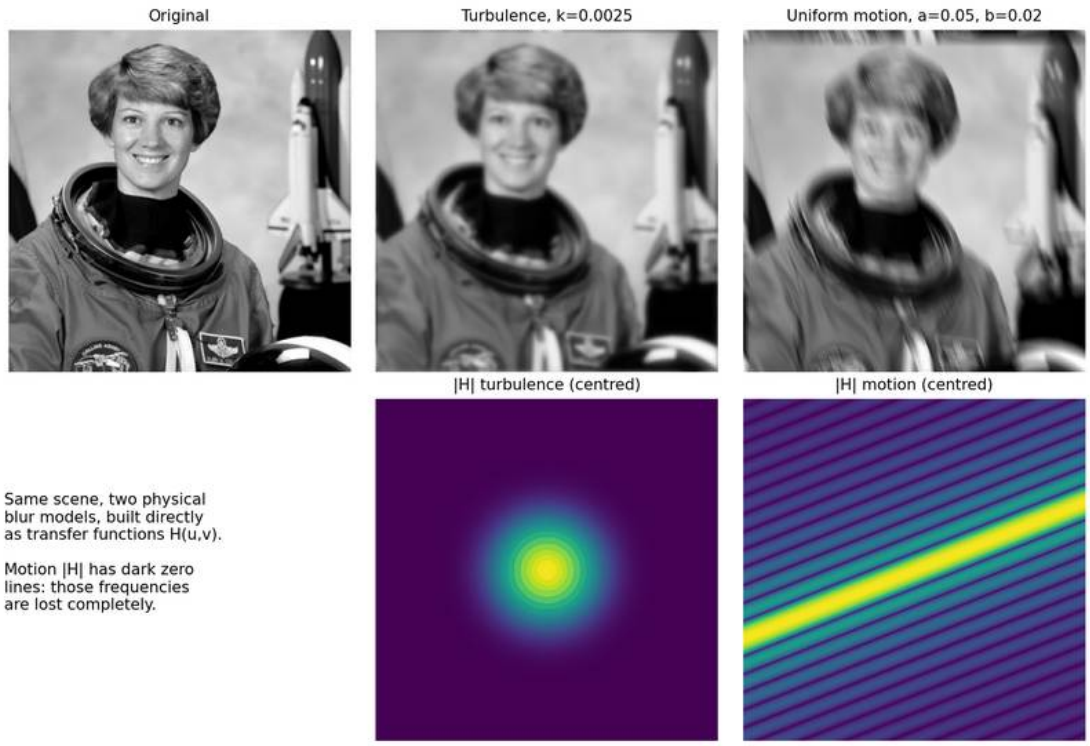


Figure 5.4 — Two degradation functions built from physics. The turbulence OTF is a smooth bump that decays but never reaches zero. The motion OTF is a sinc ridge along the motion direction with dark parallel lines of exact zeros: at those frequencies the blurred image contains no information about the scene.

```
import numpy as np

def freq_grid(shape):
    u = np.fft.fftfreq(shape[0]) * shape[0]            # 0, 1, ..., -1 (un-centred)
    v = np.fft.fftfreq(shape[1]) * shape[1]
    return np.meshgrid(u, v, indexing="ij")

def motion_blur_otf(shape, a, b, T=1.0):
    U, V = freq_grid(shape)
    s = np.pi * (U * a + V * b)
    return T * np.sinc(s / np.pi) * np.exp(-1j * s)    # np.sinc(x) = sin(pi x) / (pi x)

def turbulence_otf(shape, k):
    U, V = freq_grid(shape)
    return np.exp(-k * (U ** 2 + V ** 2) ** (5 / 6))
```

A practical detail

Building H directly in the frequency domain, as above, gives *circular* convolution: blur wraps around the image borders. That is perfect for experiments because the model matches the filter exactly. For real photos, pad the image or taper its borders before deconvolving, or you get strong ringing at the edges.

Inverse filtering

In plain words: if the blur multiplied each frequency by H, just divide by H. It is the obvious idea, and it fails spectacularly as soon as there is noise.

The **direct inverse filter** estimates

$$\hat{F}(u,v) = \frac{G(u,v)}{H(u,v)}.$$

Substituting $G = HF + N$ shows the problem:

$$\hat{F}(u,v) = F(u,v) + \frac{N(u,v)}{H(u,v)}.$$

Even if we knew H perfectly, we cannot recover F exactly, because N is unknown. Worse, blur makes $|H|$ tiny at high frequencies (or exactly zero, as for motion), while noise is roughly white and keeps its power there. So N/H becomes enormous and swamps the image. In Figure 5.5, the full inverse filter turns both images into pure noise, even though the noise standard deviation is only 0.3% and 1% of the intensity range.

A partial fix is to apply the inverse only near the origin of the spectrum, where $|H|$ is large, for instance by multiplying $\hat{F}$ by a sharp Butterworth lowpass of radius D_0. This works for the turbulence blur, whose H shrinks smoothly. It does not work well for motion blur, because the zero lines of H cross the low frequencies too. The radius must be tuned by hand. We need a method that decides *automatically* how much to trust each frequency.

Minimum mean square error (Wiener) filtering

In plain words: at every frequency, ask "is there more picture or more noise here?" Where the picture dominates, undo the blur. Where noise dominates, hold back.

The **Wiener filter** treats both the image and the noise as random fields and seeks the estimate that minimizes the mean squared error

$$e^2 = E\{(f - \hat{f})^2\},$$

where $E\{\cdot\}$ is the expected value. Assume the noise and the image are uncorrelated, that one of them has zero mean, and that $\hat{f}$ is a linear function of g. For stationary signals the DFT decouples the frequencies, so the best gain can be chosen one frequency at a time. The result is

$$\hat{F}(u,v) = \left[\frac{1}{H(u,v)}\,\frac{|H(u,v)|^2}{|H(u,v)|^2 + S_\eta(u,v)/S_f(u,v)}\right] G(u,v),$$

where $S_f(u,v) = E\{|F(u,v)|^2\}$ is the **power spectrum** of the image, $S_\eta(u,v) = E\{|N(u,v)|^2\}$ is the power spectrum of the noise, and * (used below) denotes complex conjugation.

Online only: Deriving the Wiener filter — https://blog.shyandram.dev/en/learn/image-processing/restoration-reconstruction/notes/deriving-the-wiener-filter/

The first factor is the inverse filter. The second factor is between 0 and 1. Where the signal is strong compared to the noise, $S_\eta / S_f \approx 0$, the factor is close to 1 and the filter acts as an inverse filter. Where noise dominates, the factor shrinks toward 0 and switches that frequency off. If $H = 0$ at some frequency, the bracket is 0 there (write it as $H^* / (|H|^2 + S_\eta / S_f)$ to avoid dividing by zero). With no noise, $S_\eta = 0$, the Wiener filter is the inverse filter.

The constant-K approximation

We rarely know S_f, the power spectrum of the image we are trying to find. The common shortcut replaces the ratio S_η / S_f by a single constant K:

$$\hat{F}(u, v) = \frac{H^*(u, v)}{|H(u, v)|^2 + K} G(u, v).$$

K is tuned by eye or chosen to optimize a quality measure. A useful starting point is the ratio of the noise variance to the image variance, which is the inverse of the signal-to-noise ratio. A small K gives a sharper but noisier result; a large K a smoother one.

```
import numpy as np
from skimage import data, img_as_float

rng = np.random.default_rng(0)
f = img_as_float(data.camera())[::2, ::2]                # 256 x 256
H = motion_blur_otf(f.shape, a=0.1, b=0.1)               # from the snippet above
g = np.real(np.fft.ifft2(H * np.fft.fft2(f))) + rng.normal(0, 0.01, f.shape)
G = np.fft.fft2(g)

def wiener(G, H, K):
    return np.real(np.fft.ifft2(np.conj(H) / (np.abs(H) ** 2 + K) * G))

f_hat = wiener(G, H, K=0.01)
```

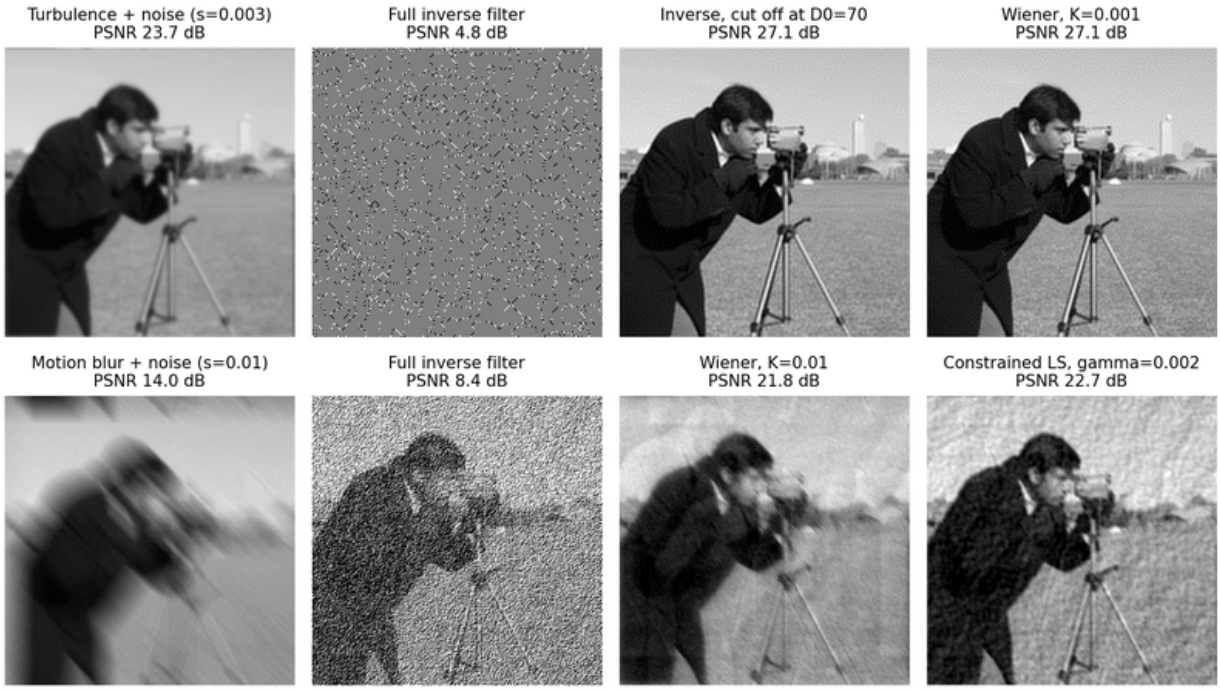


Figure 5.5 — Top row: turbulence blur ($k = 0.0025$) with weak noise. The full inverse filter is destroyed by amplified noise, the inverse restricted to a radius of 70 works, and the Wiener filter reaches the same quality without choosing a radius. Bottom row: diagonal motion blur with noise. The inverse filter fails; the Wiener and constrained least squares filters recover the scene, with some ringing and grain left over.

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

Drag the slider to compare the motion-blurred, noisy image with its Wiener restoration (full 512 × 512 version, $K = 0.01$).



Motion blur + noise



Wiener, K=0.01

Cameraman image with diagonal motion blur and noise, compared with the Wiener-filtered result

Constrained least squares filtering

In plain words: among all images that would explain what we observed, up to the expected amount of noise, pick the smoothest one.

The Wiener filter needs power spectra, or a constant K with no clear meaning. **Constrained least squares (CLS)** filtering needs only the noise mean and variance [3]. In matrix-vector form, stack the pixels into vectors so that the model is

$$\mathbf{g} = \mathbf{H}\mathbf{f} + \boldsymbol{\eta},$$

where $\mathbf{g}, \mathbf{f}, \boldsymbol{\eta}$ are $MN \times 1$ vectors and $\mathbf{H}$ is the $MN \times MN$ blur matrix. These matrices are far too large to invert directly, but because $\mathbf{H}$ represents a circular convolution it is diagonalized by the DFT, which is what makes a closed-form frequency-domain answer possible.

Inverse problems like this are **ill-conditioned**: tiny changes in $\mathbf{g}$ cause huge changes in the solution. CLS fixes this by preferring smooth solutions. It measures roughness with the Laplacian ∇^2 and solves

$$\min_{\hat{\mathbf{f}}} C = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [\nabla^2 \hat{f}(x,y)]^2 \quad \text{subject to} \quad \|\mathbf{g} - \mathbf{H}\hat{\mathbf{f}}\|^2 = \|\boldsymbol{\eta}\|^2,$$

where $\|\cdot\|$ is the Euclidean norm. The constraint says: the residual should be exactly as large as the noise, no more (we would be ignoring data) and no less (we would be fitting noise). The solution, found with a Lagrange multiplier, is

$$\hat{F}(u,v) = \left[\frac{H^*(u,v)}{|H(u,v)|^2 + \gamma\,|P(u,v)|^2} \right] G(u,v),$$

where $\gamma \geq 0$ is the multiplier and $P(u,v)$ is the DFT of the Laplacian kernel

$$p(x,y) = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix},$$

zero-padded to $M \times N$ before transforming. With $\gamma = 0$ we are back at the inverse filter. Compare with the constant-K Wiener filter: CLS replaces the flat K with $\gamma|P|^2$, which is small at low frequencies and large at high frequencies, so it holds back exactly where noise tends to dominate.

γ is not a free knob: it is chosen so that the constraint holds, that is, so that the residual $\|\mathbf{g} - \mathbf{H}\hat{\mathbf{f}}\|^2$ matches the noise energy $\|\boldsymbol{\eta}\|^2 = MN\,(\sigma_\eta^2 + \bar{\eta}^2)$, where σ_η^2 and $\bar{\eta}$ are the noise variance and mean, which we already know how to estimate from a flat patch.

Online only: Choosing the CLS multiplier — https://blog.shyandram.dev/en/learn/image-processing/restoration-reconstruction/notes/choosing-the-cls-multiplier/

```
def cls(G, H, gamma):
    p = np.zeros(H.shape)
    p[[0, 0, 0, 1, -1], [0, 1, -1, 0, 0]] = [4, -1, -1, -1, -1]   # Laplacian centred at (0,
     ↪  0)
    P = np.fft.fft2(p)
    return np.real(np.fft.ifft2(np.conj(H) / (np.abs(H) ** 2 + gamma * np.abs(P) ** 2) * G))

f_hat = cls(G, H, gamma=0.002)
```

Which is better, Wiener or CLS? Neither in general. Wiener is optimal *on average* over an ensemble of images that share the assumed power spectra. CLS is tailored to the one image at hand and can give a better result for that image when the noise statistics are known. When K or γ is hand-tuned, as is common, the two often look similar (Figure 5.5).

Geometric mean filter

In plain words: a single dial that slides between "undo the blur fully" and "play it safe against noise".

The **geometric mean filter** combines the inverse and the (parametric) Wiener filter:

$$\hat{F}(u,v) = \left[\frac{H^*(u,v)}{|H(u,v)|^2}\right]^{\alpha} \left[\frac{H^*(u,v)}{|H(u,v)|^2 + \beta \dfrac{S_\eta(u,v)}{S_f(u,v)}}\right]^{1-\alpha} G(u,v),$$

where α and β are non-negative real constants. Special cases:

- $\alpha = 1$: the inverse filter.
- $\alpha = 0$: the **parametric Wiener filter**; with $\beta = 1$ it is the standard Wiener filter.
- $\alpha = 1/2, \beta = 1$: the geometric mean of the inverse and Wiener filters, which gives this family its name. It is also called the **spectrum equalization filter**: you can check that the expected power spectrum of its output, $|W|^2(|H|^2 S_f + S_\eta)$, is then exactly S_f, the power spectrum of the original image.

With $\beta > 1$ the filter is more cautious than Wiener; with $\beta < 1$ it is bolder.

```
def geometric_mean_filter(G, H, alpha, beta, K):
    """K stands in for S_eta / S_f, as in the constant-K Wiener filter."""
    H2 = np.abs(H) ** 2
    inv = np.conj(H) / np.maximum(H2, 1e-12)
    wie = np.conj(H) / (H2 + beta * K)
    # Both factors have the phase of conj(H): combine magnitudes, keep that phase.
    return np.real(np.fft.ifft2(np.abs(inv) ** alpha * np.abs(wie) ** (1 - alpha)
                                * np.exp(1j * np.angle(wie)) * G))
```

Image reconstruction from projections

In plain words: shine X-rays through a body from many angles. Each angle gives a "shadow" that only tells how much stuff the ray passed through in total. Put enough shadows together, cleverly, and you can rebuild the inside slice.

Principles of computed tomography (CT)

In X-ray CT, a source and a row of detectors rotate around the patient. For each angle, every detector measures how much the X-ray beam was attenuated along its straight path. Attenuation along a ray adds up (the log of the intensity ratio is the line integral of the attenuation coefficient), so each measurement is a **line integral** of the unknown slice $f(x, y)$. The set of measurements for one angle is a **projection**. CT scanners evolved from a single source-detector pair that translates and rotates (parallel beams), to fan beams with many detectors, to helical and cone-beam geometries that acquire whole volumes; the reconstruction mathematics below is the core of all of them [4].

Projections and the Radon transform

Describe a straight line in normal form, $x\cos\theta + y\sin\theta = \rho$, where ρ is its distance from the origin and θ the angle of its normal. The line integral of f along that line is

$$g(\rho, \theta) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y)\, \delta(x\cos\theta + y\sin\theta - \rho)\, dx\, dy,$$

where δ is the Dirac impulse, which keeps only the points on the line. As a function of (ρ, θ), g is the **Radon transform** of f. Displayed as an image with θ on one axis and ρ on the other it is called a **sinogram**: a single point in the slice traces a sinusoid, $\rho = x_0\cos\theta + y_0\sin\theta$.

The simplest way to go back is **backprojection**: smear each projection back across the image along the direction it came from, and add up the smears over all angles:

$$f_\theta(x, y) = g(x\cos\theta + y\sin\theta,\ \theta), \qquad f_{BP}(x, y) = \int_0^{\pi} f_\theta(x, y)\, d\theta.$$

This puts mass in the right places, but also everywhere along each ray, so the result is a heavily blurred version of f (Figure 5.6, third panel). The blur is not random: it is a convolution with $1/r$, where r is the distance from the centre. The Fourier-slice theorem shows how to remove it.

The Fourier-slice theorem

Let $G(\omega, \theta)$ be the 1-D Fourier transform of the projection $g(\rho, \theta)$ with respect to ρ, at a fixed angle θ:

$$G(\omega, \theta) = \int_{-\infty}^{\infty} g(\rho, \theta)\, e^{-j2\pi\omega\rho}\, d\rho.$$

The **Fourier-slice theorem** (also called the projection-slice theorem) states that

$$G(\omega, \theta) = F(\omega \cos\theta,\ \omega \sin\theta),$$

where $F(u, v)$ is the 2-D Fourier transform of f. In words: the 1-D transform of a projection equals a radial *slice*, through the origin at angle θ, of the 2-D transform of the object. The proof is short: substitute the Radon integral into G, integrate out the delta, and the exponent becomes $-j2\pi\omega(x\cos\theta + y\sin\theta)$, which is exactly the 2-D transform evaluated at $(u, v) = (\omega\cos\theta, \omega\sin\theta)$.

So in principle, projections from all angles fill the 2-D spectrum, slice by slice, and a 2-D inverse transform returns f. In practice the slices sample the spectrum on a polar grid, densely near the origin and sparsely far from it, and interpolating onto a Cartesian grid is error-prone. That uneven sampling is also the reason plain backprojection blurs: it over-weights low frequencies.

Filtered backprojection with parallel beams

Write the 2-D inverse Fourier transform in polar coordinates $u = \omega\cos\theta$, $v = \omega\sin\theta$. The area element $du\,dv$ becomes $|\omega|\,d\omega\,d\theta$, and with the Fourier-slice theorem:

$$f(x, y) = \int_0^{\pi} \left[\int_{-\infty}^{\infty} |\omega|\, G(\omega, \theta)\, e^{j2\pi\omega\rho}\, d\omega \right]_{\rho = x\cos\theta + y\sin\theta} d\theta.$$

Read it from the inside out. For each angle, multiply the 1-D spectrum of the projection by $|\omega|$, the **ramp filter**, and transform back. Then backproject the filtered projection and sum over angles. This is **filtered backprojection (FBP)**. The ramp exactly cancels the $1/r$ blur of plain backprojection.

The ramp grows without bound, so it amplifies high-frequency noise and is not integrable. In practice it is band-limited and multiplied by a smooth window, for example a Hamming window $h(\omega) = c + (1 - c)\cos(2\pi\omega/W)$ inside the band $|\omega| \le W/2$ and 0 outside, where W is the bandwidth and $c = 0.54$. The window equals 1 at $\omega = 0$ and falls to $2c - 1 = 0.08$ at the band edge. The window trades a little sharpness for much less ringing and noise. A filter introduced alongside the famous head phantom of Shepp and Logan [5] is still one of the standard options. The number of views matters as well: with too few angles, FBP produces streaks (Figure 5.6, last panel).

```
import numpy as np
from skimage.data import shepp_logan_phantom
from skimage.transform import radon, iradon, rescale

f = rescale(shepp_logan_phantom(), 0.64)                 # 256 x 256
theta = np.linspace(0.0, 180.0, 180, endpoint=False)     # projection angles in degrees
sinogram = radon(f, theta=theta)                         # shape: (detector bins, angles)
backproj = iradon(sinogram, theta=theta, filter_name=None)    # blurry, no filter
fbp = iradon(sinogram, theta=theta, filter_name="ramp")       # filtered backprojection
rmse = np.sqrt(np.mean((fbp - f) ** 2))
```

`iradon` also accepts `"shepp-logan"`, `"cosine"`, `"hamming"` and `"hann"` windows, and `skimage.transform.iradon_sart` gives an iterative alternative [6].

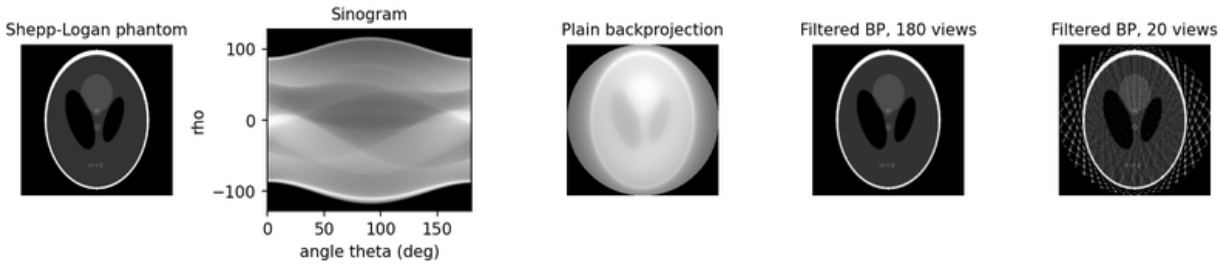


Figure 5.6 — From slice to sinogram and back. Each point of the phantom traces a sinusoid in the sinogram. Plain backprojection gives a blurred halo; adding the ramp filter (FBP) recovers the slice. With only 20 views, FBP shows streak artifacts, which is the problem learned reconstruction methods address (see Modern view).

Online only: Fan-beam filtered backprojection — https://blog.shyandram.dev/en/learn/image-processing/restoration-reconstruction/notes/fan-beam-filtered-backprojection/

Modern view

The tools of this chapter are not museum pieces. The model $g = h \star f + \eta$ is still how the field states its problems, and Wiener-like steps still sit inside modern networks. What deep learning changed is the *prior*: instead of assuming "the image is smooth" (CLS) or "the image has power spectrum S_f" (Wiener), we now learn what clean images look like from data. For a broad tour of deep image restoration beyond this chapter (super-resolution, dehazing, deraining, low-light, all-in-one restoration and diffusion priors), see DLCV 09 ·Deep Image Restoration. The surveys below focus on the three topics of this chapter: denoising, deblurring and CT reconstruction.

Online only: The Modern view surveys in more detail — https://blog.shyandram.dev/en/learn/image-processing/restoration-reconstruction/papers/the-modern-view-surveys-in-more-detail/

Denoising: from patches to networks

Two classical methods marked the peak of hand-designed denoising. **Non-local means (NLM)** [7] replaces each pixel by a weighted average of pixels whose *surrounding patches* look similar, wherever they are in the image: the adaptive filter idea taken much further. **BM3D** [8] groups similar patches into 3-D stacks, filters each stack jointly in a transform domain and finishes with a Wiener-type step. It was the standard baseline for about a decade.

Online only: Non-local means denoising — https://blog.shyandram.dev/en/learn/image-processing/restoration-reconstruction/notes/non-local-means-denoising/

Online only: Non-Local Means: use similar patches to decide which pixels to average — https://blog.shyandram.dev/en/learn/image-processing/restoration-reconstruction/papers/non-local-means-use-similar-patches-to-decide-which-pixels-to-average/; BM3D: denoise groups of similar patches collaboratively — https://blog.shyandram.dev/en/learn/image-processing/restoration-reconstruction/papers/bm3d-denoise-groups-of-similar-patches-collaboratively/

DnCNN [9] showed that a plain deep convolutional network trained to predict the noise (residual learning) beats BM3D on Gaussian noise, and that one model can handle unknown noise levels. Two surveys map what followed:

- **Tian et al. (2020)** [10] sorts deep denoisers by noise type: synthetic Gaussian noise, real camera noise, unknown (blind) noise and mixed cases. The noise model of this chapter is still the organizing idea.
- **Elad, Kawar and Vaksman (2023)** [11] argues that denoisers now matter far beyond denoising: a good denoiser serves as the prior for other inverse problems and is the engine of diffusion-based image generation.

Deblurring and general inverse problems

- **Zhang et al. (IJCV 2022)** [12] surveys deep deblurring. With a known blur, as in this chapter, the work is mostly about suppressing the noise amplification and ringing of the inverse filter; blind deblurring, which must also estimate h, remains the harder problem.
- **Ongie et al. (2020)** [13] sorts deep methods for $\mathbf{g} = \mathbf{H}\mathbf{f} + \boldsymbol{\eta}$ by how much is known about $\mathbf{H}$ and how they are trained. How much you know about $\mathbf{H}$ is the first design decision.
- **Kamilov et al. (2023)** [14] reviews plug-and-play (PnP) methods, which alternate a physics step ($\mathbf{g} \approx \mathbf{H}\mathbf{f}$) with a call to a denoiser that acts as the prior. DPIR [15] is a widely used example.

The connection to this chapter is direct. For a blur H, the physics step of such a splitting method asks for $\hat{F}$ minimizing $|G - HF|^2 + \mu|F - Z|^2$ at each frequency, where Z is the current denoised estimate and μ a penalty weight. Its solution is $\hat{F} = (H^*G + \mu Z)/(|H|^2 + \mu)$, which is a constant-$K$ Wiener filter pulled toward Z instead of toward zero.

CT reconstruction: learned and unrolled

FBP is fast and exact for complete, low-noise data. It struggles when data are reduced to lower the radiation dose: fewer views give streaks (Figure 5.6), fewer photons per view give noise. Classical answers are iterative methods with hand-made priors such as total variation (*optional: Total variation as a regularizer* (*https://blog.shyandram.dev/en/learn/image-processing/restoration-reconstruction/notes/total-variation-as-a-regularizer/*), *in the Supplementary notes*). Deep learning adds two main patterns, reviewed for CT and other tomographic modalities by Wang, Ye and De Man [19]:

- **Post-processing:** run FBP, then a CNN removes the artifacts, as in **FBPConvNet** [16].
- **Unrolled reconstruction:** fix the number of iterations of an iterative algorithm and replace parts of each iteration with small trained networks, as in **Learned Primal-Dual** [17]. Monga, Li and Eldar [18] survey this design pattern.

What did *not* change: the Radon transform and the Fourier-slice theorem define the forward model that every one of these methods uses, and FBP remains the default in clinical scanners and the input or starting point of many learned methods. Learned methods also bring a new risk

that this chapter's linear filters do not have: a network can produce plausible structure that is not in the data. That is why validation on real, out-of-distribution scans matters in medical use.

Key takeaways

- Restoration is objective: model the degradation as $g = h \star f + \eta$ (or $G = HF + N$), then invert *that* model.
- Identify noise from the histogram of a flat patch; estimate its mean and variance from the same patch.
- Match the spatial filter to the noise: means for Gaussian-like noise, contraharmonic with the right sign of Q for one-sided impulses, median and adaptive median for salt-and-pepper, alpha-trimmed for mixtures.
- Periodic noise is a few spikes in the spectrum; notch filters remove it, and optimum notch filtering adapts the subtraction locally.
- The inverse filter G/H amplifies noise wherever $|H|$ is small. Wiener, constrained least squares and geometric mean filters all add a term to the denominator that holds back those frequencies.
- CT measures line integrals (the Radon transform). By the Fourier-slice theorem, each projection gives one slice of the 2-D spectrum, and filtered backprojection (ramp filter, then backproject) reconstructs the image.
- Deep learning replaced hand-made priors with learned ones, but the best methods still keep the forward model inside (plug-and-play, unrolled networks).

Exercises

1. A dark-field microscope image has a background region whose histogram is one-sided, starting sharply at 12 and with a long right tail; its mean is 20 and its variance is 64 (gray levels). Which PDF from this chapter would you try first, and what are its parameters?

   Hint

   A sharp start and a long tail point to the exponential, Rayleigh or Erlang family. Try an exponential shifted to start at 12: it needs $1/a = 20 - 12 = 8$ and $1/a^2 = 64$, and both hold with $a = 1/8$. A Rayleigh fit is inconsistent: $b(4 - \pi)/4 = 64$ gives $b \approx 298$, so its start would be $20 - \sqrt{\pi b/4} \approx 4.7$, not 12.

2. An image is corrupted only by salt noise. Without testing, predict what happens with (a) a max filter, (b) a min filter, (c) a contraharmonic filter with $Q = 2$, (d) with $Q = -2$. Then verify with the code in this chapter.

   Hint

   Salt pixels are the largest values in their windows. The max filter spreads them (worse), the min filter removes them (but darkens and erodes bright detail). Positive Q weights large values more, so $Q = 2$ makes salt worse; $Q = -2$ removes it.

3. Show that the Wiener filter with constant K reduces to the inverse filter when $K = 0$, and that for a frequency where $|H|^2 \ll K$ the output is approximately $(H^*/K)\,G$. What does this say about how Wiener filtering treats frequencies the blur destroyed?

Hint

When $|H|^2 \ll K$ the denominator is about K. The output is then small, proportional to $|H|$, so the filter does *not* try to recover those frequencies. It gives up on them gracefully instead of amplifying noise.

4. A 256×256 image is blurred by uniform motion of 8 pixels along x and none along y. Frequencies are integers (cycles per image), as in the code above. Using the motion blur OTF, find all frequencies (u, v) with $H(u, v) = 0$. Explain why no linear filter can restore detail at those frequencies, and what that looks like in the restored image.

Hint

With u in cycles per image, a is the blur length as a fraction of the image size, so $a = 8/256 = 1/32$ and $b = 0$. Then $H = 0$ when $u/32$ is a non-zero integer: $u = \pm 32, \pm 64, \pm 96, \pm 128$, for every v. Since $G = HF + N$ has no F content there, any linear filter outputs only noise (or zero) at those frequencies. Expect faint ringing at that spatial period along x.

5. In the CT code above, reconstruct with 180, 60, 20 and 10 views, and with the `ramp` and `hann` filters. Plot RMSE versus number of views. Then add Gaussian noise to the sinogram and repeat. When does the smoother `hann` filter win?

Hint

Without noise, `ramp` should win at many views. As noise grows, the ramp amplifies it at high frequencies and `hann` (which rolls off there) wins, which is the same trade-off as inverse filter versus Wiener filter.

6. Derive the per-frequency solution $\hat{F} = (H^*G + \mu Z)/(|H|^2 + \mu)$ of $\min_F |G - HF|^2 + \mu|F - Z|^2$. What filter from this chapter do you get when $Z = 0$?

Hint

Differentiate with respect to F^*: $-H^*(G - HF) + \mu(F - Z) = 0$. With $Z = 0$ it is the Wiener filter with constant $K = \mu$.

References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018, Ch. 5. publisher page
2. H. Hwang and R. A. Haddad, "Adaptive median filters: new algorithms and results," *IEEE Transactions on Image Processing*, vol. 4, no. 4, pp. 499–502, 1995. DOI

3. B. R. Hunt, “The application of constrained least squares estimation to image restoration by digital computer,” *IEEE Transactions on Computers*, vol. C-22, no. 9, pp. 805–812, 1973. DOI
4. A. C. Kak and M. Slaney, *Principles of Computerized Tomographic Imaging*, IEEE Press, 1988 (SIAM reprint 2001). free online edition
5. L. A. Shepp and B. F. Logan, “The Fourier reconstruction of a head section,” *IEEE Transactions on Nuclear Science*, vol. 21, no. 3, pp. 21–43, 1974. DOI
6. scikit-image developers, “Radon transform,” scikit-image documentation (gallery example). docs
7. A. Buades, B. Coll and J.-M. Morel, “A non-local algorithm for image denoising,” *IEEE CVPR*, vol. 2, pp. 60–65, 2005. DOI
8. K. Dabov, A. Foi, V. Katkovnik and K. Egiazarian, “Image denoising by sparse 3-D transform-domain collaborative filtering,” *IEEE Transactions on Image Processing*, vol. 16, no. 8, pp. 2080–2095, 2007. DOI
9. K. Zhang, W. Zuo, Y. Chen, D. Meng and L. Zhang, “Beyond a Gaussian denoiser: Residual learning of deep CNN for image denoising,” *IEEE Transactions on Image Processing*, vol. 26, no. 7, pp. 3142–3155, 2017. doi ·arXiv
10. C. Tian, L. Fei, W. Zheng, Y. Xu, W. Zuo and C.-W. Lin, “Deep learning on image denoising: An overview,” *Neural Networks*, vol. 131, pp. 251–275, 2020. arXiv
11. M. Elad, B. Kawar and G. Vaksman, “Image denoising: The deep learning revolution and beyond — A survey paper,” *SIAM Journal on Imaging Sciences*, vol. 16, no. 3, pp. 1594–1654, 2023. arXiv
12. K. Zhang, W. Ren, W. Luo, W.-S. Lai, B. Stenger, M.-H. Yang and H. Li, “Deep image deblurring: A survey,” *International Journal of Computer Vision*, 2022. arXiv
13. G. Ongie, A. Jalal, C. A. Metzler, R. G. Baraniuk, A. G. Dimakis and R. Willett, “Deep learning techniques for inverse problems in imaging,” arXiv:2005.06001, 2020. arXiv
14. U. S. Kamilov, C. A. Bouman, G. T. Buzzard and B. Wohlberg, “Plug-and-play methods for integrating physical and learned models in computational imaging: Theory, algorithms, and applications,” *IEEE Signal Processing Magazine*, vol. 40, no. 1, pp. 85–97, 2023. arXiv
15. K. Zhang, Y. Li, W. Zuo, L. Zhang, L. Van Gool and R. Timofte, “Plug-and-play image restoration with deep denoiser prior,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 10, pp. 6360–6376, 2022. arXiv
16. K. H. Jin, M. T. McCann, E. Froustey and M. Unser, “Deep convolutional neural network for inverse problems in imaging,” *IEEE Transactions on Image Processing*, vol. 26, no. 9, pp. 4509–4522, 2017. arXiv
17. J. Adler and O. Öktem, “Learned primal-dual reconstruction,” *IEEE Transactions on Medical Imaging*, 2018. arXiv
18. V. Monga, Y. Li and Y. C. Eldar, “Algorithm unrolling: Interpretable, efficient deep learning for signal and image processing,” arXiv:1912.10557, 2019. arXiv
19. G. Wang, J. C. Ye and B. De Man, “Deep learning for tomographic image reconstruction,” *Nature Machine Intelligence*, vol. 2, no. 12, pp. 737–748, 2020. DOI
20. A. Buades, B. Coll and J.-M. Morel, “A review of image denoising algorithms, with a new one,” *Multiscale Modeling & Simulation*, vol. 4, no. 2, pp. 490–530, 2005. DOI
21. L. I. Rudin, S. Osher and E. Fatemi, “Nonlinear total variation based noise removal algorithms,” *Physica D: Nonlinear Phenomena*, vol. 60, no. 1–4, pp. 259–268, 1992. DOI

22. A. Chambolle, “An algorithm for total variation minimization and applications,” *Journal of Mathematical Imaging and Vision*, vol. 20, no. 1–2, pp. 89–97, 2004. DOI
23. A. Buades, B. Coll and J.-M. Morel, “Non-Local Means Denoising,” *Image Processing On Line*, vol. 1, pp. 208–212, 2011. DOI ·article and implementation

IP 06 • Color Image Processing

How color is measured and encoded, how to convert between RGB, HSI and L*a*b*, and how to transform, filter and segment color images.

Online edition: https://blog.shyandram.dev/en/learn/image-processing/color/

What you'll learn

- Why three numbers are enough to describe a color for a human observer, and what the CIE chromaticity diagram and a device gamut are
- How the RGB, CMY(K), HSI and CIE L*a*b* models relate, with exact conversion formulas, plus where sRGB gamma and YCbCr fit in
- How pseudocolor turns a grayscale image into a readable color picture, and why perceptually uniform colormaps such as viridis beat the rainbow
- How to transform, white-balance, equalize, smooth, sharpen and segment full-color images, and when to process channels separately versus as vectors
- How noise behaves in color images and how vector filters remove it
- What modern research (learning-based color constancy, colormap design, deep colorization) adds to the classical toolkit

The big picture

So far every pixel has been one number: its brightness. Color images store three numbers per pixel. Color helps us find objects ("the red car") and separate materials, and it is the first thing people notice when a photo looks wrong.

> **Explain like I'm five**
>
> Imagine you have only three flashlights: a red one, a green one and a blue one. Shine them on a white wall. If you turn them up and down, you can make almost any color you like: red plus green makes yellow, all three make white. Your eyes work the same way in reverse: they have three kinds of tiny color sensors, and your brain mixes the three signals into every color you see. A color image is just a note, for every dot, of how bright each of the three flashlights should be.

Following the textbook [1], the chapter answers three questions: how to describe color precisely (fundamentals and models), how to use color to show data that is not colorful (pseudocolor), and how to process real color photographs (transformations, filtering, segmentation, noise, compression). Most grayscale tools from Chapters 3 to 5 carry over, with one recurring twist: a color pixel is a **vector**, and treating it as three unrelated numbers is sometimes fine and sometimes badly wrong.

Color fundamentals

In plain words. Light is a mix of wavelengths. Our eyes do not measure that whole mix. They squash it into three numbers, one per type of color sensor. Two very different light mixes that produce the same three numbers look identical to us.

Light, cones and trichromacy

Visible light spans wavelengths of roughly 400 to 700 nanometers. A light source or a surface is described by its **spectral power distribution** $S(\lambda)$: how much energy it emits or reflects at each wavelength λ.

The retina has three kinds of **cones** (color-sensitive photoreceptors), most sensitive to short, medium and long wavelengths. Each cone type $c \in \{S, M, L\}$ has a sensitivity curve $s_c(\lambda)$, and its response is

$$r_c = \int S(\lambda)\, s_c(\lambda)\, d\lambda.$$

Here $S(\lambda)$ is the incoming spectrum and $s_c(\lambda)$ the sensitivity of cone type c. Because a whole spectrum is reduced to three numbers, human color vision is **trichromatic**. A direct consequence is **metamerism**: two different spectra $S_1 \neq S_2$ that give the same three responses look like the same color. Metamerism is why displays work: a monitor never reproduces a lemon's spectrum, only its three cone responses.

Brightness, hue and saturation

People describe a color with three perceptual attributes:

- **Brightness**: how much light seems to be there (for a colorless gray, this is all there is).
- **Hue**: the dominant wavelength, the thing we name with words such as red, orange or blue.
- **Saturation**: how pure the color is, i.e. how little white is mixed in. Pink is a low-saturation red.

Hue and saturation together are called **chromaticity**, so a color is "chromaticity plus brightness", the idea behind the HSI model.

CIE tristimulus values and the chromaticity diagram

To make color measurable, the Commission Internationale de l'Éclairage (CIE) standardized, in 1931, three **color-matching functions** $\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda)$ derived from human matching experiments. The **tristimulus values** of a spectrum are

$$X = \int S(\lambda)\, \bar{x}(\lambda)\, d\lambda, \qquad Y = \int S(\lambda)\, \bar{y}(\lambda)\, d\lambda, \qquad Z = \int S(\lambda)\, \bar{z}(\lambda)\, d\lambda.$$

X, Y, Z are non-negative by construction, and $\bar{y}$ was chosen to match the eye's brightness sensitivity, so Y is the **luminance**. To separate chromaticity from brightness, normalize:

$$x = \frac{X}{X+Y+Z}, \qquad y = \frac{Y}{X+Y+Z}, \qquad z = \frac{Z}{X+Y+Z} = 1 - x - y.$$

x, y are the **chromaticity coordinates**; z is redundant. Plotting every pure spectral color (a single wavelength) in the (x, y) plane traces a horseshoe-shaped curve called the **spectral locus**. The straight line joining its two ends is the **line of purples** (purples are not single wavelengths). Every color a human can see lies inside this region.

Two geometric facts make the diagram useful:

1. **Mixing is linear.** Any mixture of two lights lies on the straight segment between their chromaticities. Any mixture of three lights lies inside their triangle.
2. **White sits in the middle.** A reference white, for example the daylight illuminant D65 at $(x, y) \approx (0.3127, 0.3290)$, lies near the center. Moving from white toward the locus increases saturation; the direction gives the hue.

Gamut

A display with three primaries can only produce the colors inside the triangle formed by those primaries. That triangle is the device's **gamut**. Because the spectral locus is curved, no triangle with physically realizable corners can cover it. Every three-primary display therefore misses some visible colors, especially saturated cyans and greens.

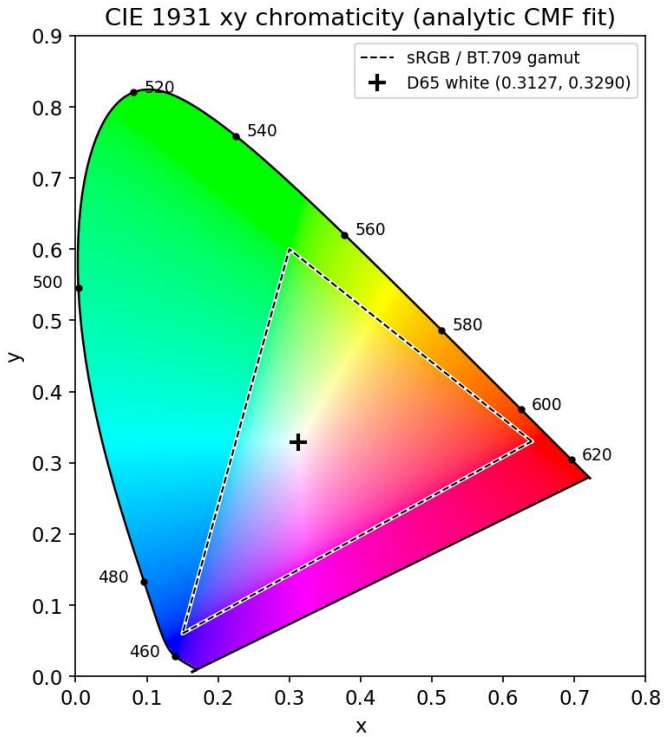


Figure 6.1 — CIE 1931 xy chromaticity diagram computed from the analytic color-matching-function fit

of Wyman et al. [2] (430–650 nm; the far tails of the fit are omitted). The dashed triangle is the sRGB / BT.709 gamut [3], [4]; colors outside it are only approximated on your screen.

Online only: The analytic color-matching fit behind Figure 6.1 — https://blog.shyandram.dev/en/learn/image-processing/color/notes/the-analytic-color-matching-fit-behind-figure-61/

Color models

In plain words. A color model is a coordinate system for colors. Different jobs need different coordinates: screens think in red-green-blue light, printers in inks, and people in "which color, how vivid, how bright".

RGB

In the **RGB model** each color is a point (R, G, B) in a unit cube. Black is $(0, 0, 0)$, white is $(1, 1, 1)$, and the main diagonal holds the grays. An 8-bit-per-channel image has $2^{24} \approx 16.7$ million possible colors. RGB is **additive**: it describes light being added together, which is how displays and camera sensors work.

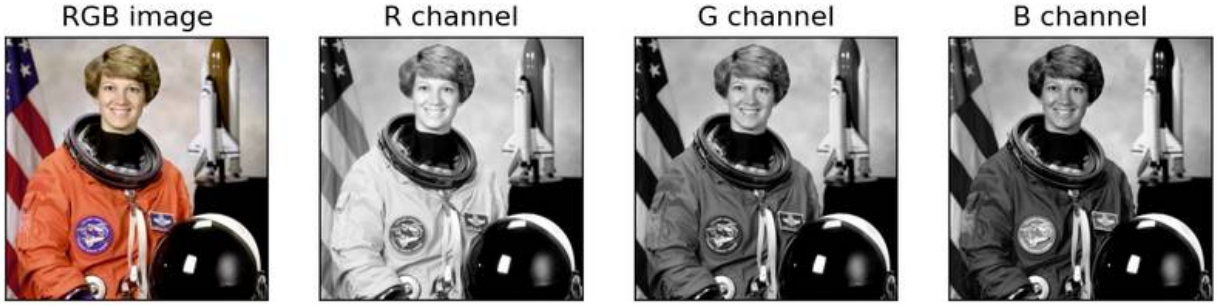


Figure 6.2 — An RGB image is three grayscale images stacked together. The orange suit is bright in R and dark in B; the white parts of the suit are bright in all three.

"RGB" alone is not a precise color: $(1, 0, 0)$ means a different red on every display. The **sRGB** standard [3] fixes this by specifying primaries (the same chromaticities as ITU-R BT.709 [4]), the D65 white point, and a nonlinear encoding. The encoding, often loosely called "gamma", converts a stored value $C' \in [0, 1]$ to linear light C:

$$C = \begin{cases} C'/12.92, & C' \le 0.04045, \\ \left(\dfrac{C' + 0.055}{1.055}\right)^{2.4}, & C' > 0.04045. \end{cases}$$

C' is the stored (encoded) value of one channel and C the linear-light value. The curve spends more code values on dark tones, where the eye is more sensitive. **Practical rule:** operations that model physics (blurring, resizing, mixing light) are more correct on linear values, so decode first when accuracy matters.

CMY and CMYK

Printing works the other way round. Ink on white paper *removes* light, so printing is **subtractive**. The secondary colors cyan, magenta and yellow absorb red, green and blue respectively. With values in $[0,1]$:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}.$$

Here C, M, Y are ink amounts (not to be confused with the CIE Y). Equal amounts of real C, M and Y inks give a muddy brown, not black, so printers add a fourth black ink K. A simple conversion is

$$K = \min(C, M, Y), \qquad C_K = \frac{C - K}{1 - K}, \quad M_K = \frac{M - K}{1 - K}, \quad Y_K = \frac{Y - K}{1 - K} \quad (K < 1),$$

with $C_K = M_K = Y_K = 0$ when $K = 1$ (pure black). Real printers use measured profiles instead, but the idea is the same: move the shared gray part into black ink.

HSI

RGB is convenient for hardware but awkward for people: nobody describes a color as "70% red, 40% green, 20% blue". The **HSI model** (hue, saturation, intensity) decouples brightness from chromaticity, which matches how we talk about color and makes many algorithms simpler.

Geometrically, stand the RGB cube on its black corner so the gray diagonal points up. **Intensity** is the height along that axis. Cut the cube with a plane perpendicular to the axis: **hue** is the angle around the axis (red at 0°, green at 120°, blue at 240°), and **saturation** is the distance from the axis.

RGB to HSI (with $R, G, B \in [0,1]$):

$$\theta = \cos^{-1}\left(\frac{\frac{1}{2}[(R-G) + (R-B)]}{\sqrt{(R-G)^2 + (R-B)(G-B)}} \right), \qquad H = \begin{cases} \theta, & B \le G \\ 360^\circ - \theta, & B > G \end{cases}$$

$$S = 1 - \frac{3\,\min(R,G,B)}{R+G+B}, \qquad I = \frac{R+G+B}{3}.$$

θ is the angle between the pixel's chromatic component and the red axis. H is hue in degrees, $S \in [0,1]$ saturation, $I \in [0,1]$ intensity. Two edge cases need care in code. For gray pixels ($R = G = B$) the denominator of θ is zero and hue is **undefined**; set it to 0 by convention. For black ($I = 0$) saturation is undefined; set it to 0 as well.

HSI to RGB inverts these formulas. The conversion is exact, but it works sector by sector: which formula applies depends on which third of the color circle (0°–120°, 120°–240° or 240°–360°) the hue falls in, and the channel "opposite" the hue always gets the smallest value $I(1 - S)$.

Online only: Converting HSI back to RGB — https://blog.shyandram.dev/en/learn/image-processing/color/notes/converting-hsi-back-to-rgb/

```
import numpy as np

def rgb_to_hsi(rgb):
    """rgb: float array in [0, 1] with shape (..., 3). Returns H (degrees), S, I."""
    R, G, B = rgb[..., 0], rgb[..., 1], rgb[..., 2]
    num = 0.5 * ((R - G) + (R - B))
    den = np.sqrt((R - G) ** 2 + (R - B) * (G - B))
    theta = np.degrees(np.arccos(np.clip(num / (den + 1e-12), -1, 1)))
    H = np.where(B > G, 360.0 - theta, theta)
    I = rgb.mean(axis=-1)
    S = np.where(I > 0, 1 - rgb.min(axis=-1) / (I + 1e-12), 0.0)
    H = np.where(den > 1e-12, H, 0.0)      # gray pixels: hue undefined -> 0
    return H, S, I

prim = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1], [0.5, 0.5, 0.5]], float)
print(np.round(np.stack(rgb_to_hsi(prim), -1), 3))
# [[  0.   1.   0.333] [120.  1.  0.333] [240.  1.  0.333] [  0.  0.  0.5 ]]
```

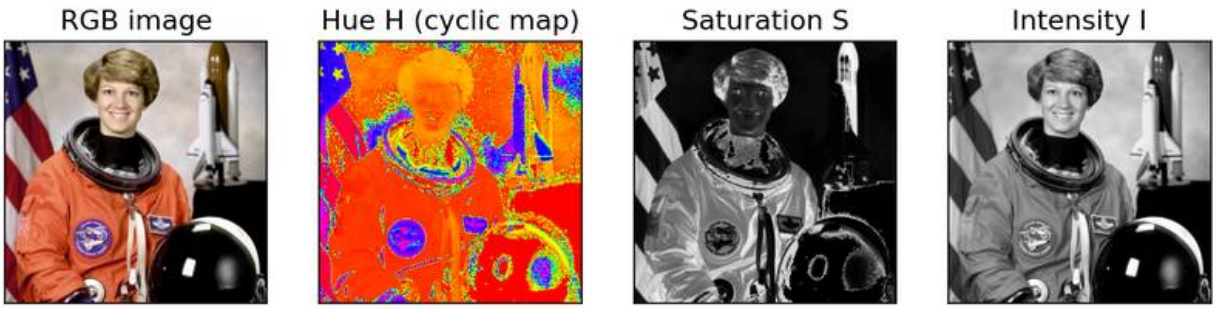


Figure 6.3 — HSI components. Hue is shown with a cyclic colormap because 0° and 360° are the same hue. Hue is noisy in gray regions (background, white suit), where it is nearly undefined; saturation is high only on the orange suit, the flag and the patches.

> **HSI, HSV and HSL are cousins, not twins**
>
> Libraries usually implement HSV (value $= \max(R,G,B)$) or HSL, not the HSI formulas above. All three separate hue from the rest, but their saturation and brightness axes differ, so numbers are not interchangeable. OpenCV also stores 8-bit hue as $H/2$ so that it fits in $[0, 179]$.

CIE L*a*b*

HSI separates brightness from chromaticity, but equal steps in HSI do not look equally large. The **CIE 1976 L*a*b*** space [5] was designed so that Euclidean distance roughly tracks perceived difference. It has three axes:

- $L^* \in [0, 100]$ is **lightness**, from black to white;
- a^* runs from green (negative) to red (positive);

- b^* runs from blue (negative) to yellow (positive).

It is computed from the tristimulus values X, Y, Z divided by those of a reference white, followed by a cube root that mimics the eye's compressive response. The color difference ΔE^*_{ab} is simply the Euclidean distance between two (L^*, a^*, b^*) points. In code, `skimage.color.rgb2lab` converts an sRGB image in one call.

Online only: CIELAB formulas and a from-scratch conversion — https://blog.shyandram.dev/en/learn/image-processing/color/notes/cielab-formulas-and-a-from-scratch-conversion/

YCbCr in one paragraph

Video and image codecs separate a **luma** signal Y' from two **chroma** differences, because the eye resolves fine detail in brightness much better than in color. ITU-R BT.601 [6] defines, for gamma-encoded $R', G', B' \in [0, 1]$,

$$Y' = 0.299\, R' + 0.587\, G' + 0.114\, B',$$

with $C_b \propto B' - Y'$ and $C_r \propto R' - Y'$ scaled and offset into 8-bit "studio" ranges (Y' in $[16, 235]$, C_b, C_r in $[16, 240]$). HDTV uses BT.709 [4], whose luma weights are different $(0.2126, 0.7152, 0.0722)$. Mixing them up shifts colors slightly but visibly. `skimage.color.rgb2ycbcr` implements the BT.601 studio-range version.

Pseudocolor image processing

In plain words. Pseudocolor (false color) paints a grayscale image with colors so that small differences in brightness become easy to see. The colors are not "real"; they are a code, like the colors on a weather map.

The motivation is perceptual: people can tell apart far more color combinations than shades of gray. A thermal camera, an X-ray, a depth map or an elevation model only has one value per pixel, but rendering that value with color can reveal structure the eye would miss in gray.

Intensity slicing

The simplest scheme cuts the gray range into bands and gives each band a color. With thresholds $0 = l_0 < l_1 < \cdots < l_P = L - 1$ and colors $c_1, \ldots, c_P$,

$$g(x, y) = c_k \quad \text{if} \quad l_{k-1} \le f(x, y) < l_k.$$

f is the input gray level, L the number of gray levels, P the number of bands, and g the output color. Viewed as a 3-D plot of f, each threshold is a horizontal plane slicing the "intensity terrain", hence the name. With $P = 2$ this is plain thresholding drawn in two colors. Intensity slicing is ideal when the bands mean something: "temperature above 40 °C", "bone versus soft tissue".

Intensity-to-color transformations

A more general approach applies three independent functions to the gray value:

$$g_R(x,y) = T_R(f(x,y)), \qquad g_G(x,y) = T_G(f(x,y)), \qquad g_B(x,y) = T_B(f(x,y)).$$

T_R, T_G, T_B map gray level to the red, green and blue output. Intensity slicing is the special case where all three are staircases. Smooth functions give a **continuous colormap**. The same idea extends to several input images: for multispectral data, three bands (for example, near-infrared, red and green) are each assigned to a display channel to form a false-color composite.

Perceptually uniform colormaps

The choice of T_R, T_G, T_B matters more than it looks. The popular rainbow map (`jet`) passes through bright yellow and cyan in the middle and gets darker at both ends. Its **lightness is not monotonic**: equal steps in data produce unequal visible steps, flat bands look like edges, and real edges can vanish. Kovesi [7] shows that some widely used maps have perceptual "flat spots" that can hide a feature as large as a tenth of the data range, and argues that a colormap should be designed so that lightness changes uniformly.

viridis, designed by van der Walt and Smith and now Matplotlib's default [8], was built to be perceptually uniform in the CAM02-UCS color-appearance space, to stay uniform when printed in grayscale, and to remain readable for common forms of color-vision deficiency. Crameri, Shephard and Heron [9] make the same case across the sciences: rainbow and red–green maps distort data and exclude the roughly 8% of men with color-vision deficiency.

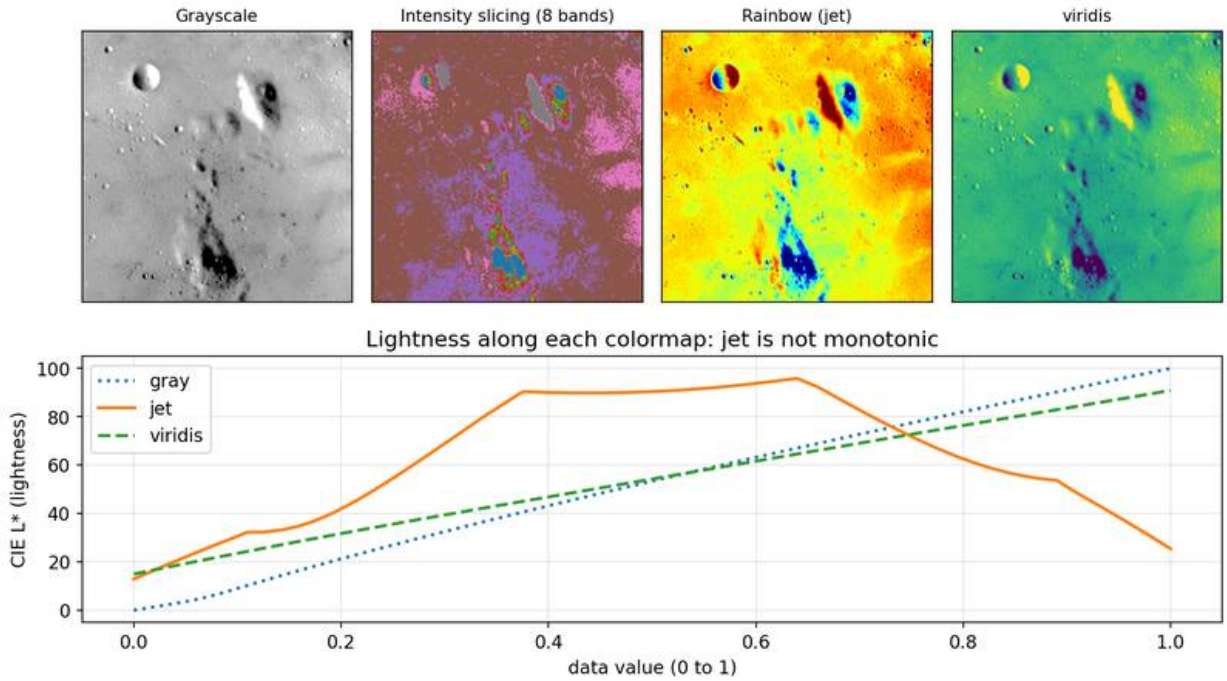


Figure 6.4 — Pseudocolor of a contrast-stretched lunar image. Top: grayscale, intensity slicing into 8 bands, the rainbow map jet, and viridis. Bottom: L lightness along each map. viridis rises steadily like gray; jet rises, plateaus and falls, which creates false edges.*

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import data, util, color

g = util.img_as_float(data.moon())
# intensity slicing into 8 equal bands
bands = np.digitize(g, np.linspace(0, 1, 9)[1:-1])          # 0..7
palette = plt.get_cmap("tab10")(np.arange(8))[:, :3]
sliced = palette[bands]                                      # (H, W, 3)
# continuous intensity-to-color transformation
rgb = plt.get_cmap("viridis")(g)[..., :3]
# check a colormap's lightness profile
L = color.rgb2lab(plt.get_cmap("jet")(np.linspace(0, 1, 256))[None, :, :3])[0, :, 0]
print(np.all(np.diff(L) >= 0))                               # False: jet is not monotonic
```

Rule of thumb: use a **sequential** uniform map (such as viridis) for ordered data, a **diverging** map when a meaningful midpoint exists (zero, a threshold), a **cyclic** map for angles and hue, and a **qualitative** palette only for unordered categories, as in intensity slicing.

Basics of full-color image processing

In plain words. You can process a color image as three separate gray images, or treat each pixel as a little arrow in color space and process the arrows. Sometimes both give the same answer; often they do not.

A full-color pixel is the vector

$$\mathbf{c}(x,y) = \begin{bmatrix} c_R(x,y) \\ c_G(x,y) \\ c_B(x,y) \end{bmatrix} = \begin{bmatrix} R(x,y) \\ G(x,y) \\ B(x,y) \end{bmatrix}.$$

There are two ways to process it.

1. **Per-channel (componentwise).** Apply a grayscale method to each channel independently and restack.
2. **Vector processing.** Work directly on the vectors $\mathbf{c}(x,y)$ using operations that treat the three components together, such as vector distances, vector medians or the color gradient.

Per-channel processing gives the same result as vector processing when two conditions hold: the operation is applicable to both scalars and vectors, and it acts on each component in the same way without mixing them. **Linear** neighborhood operations satisfy both. Averaging each channel over a window equals averaging the vectors over that window:

$$\frac{1}{|S_{xy}|}\sum_{(s,t)\in S_{xy}}\mathbf{c}(s,t)=\begin{bmatrix}\frac{1}{|S_{xy}|}\sum R(s,t)\\ \frac{1}{|S_{xy}|}\sum G(s,t)\\ \frac{1}{|S_{xy}|}\sum B(s,t)\end{bmatrix},$$

where S_{xy} is the neighborhood centered at (x,y) and $|S_{xy}|$ its number of pixels. **Nonlinear** operations break the equivalence. A per-channel median can pick the red value from one pixel, the green from a second and the blue from a third, creating a color that appeared nowhere in the neighborhood. Thresholds, edge detectors and histogram equalization are other cases where the vector view matters, as later sections show.

Color transformations

In plain words. A color transformation is a recipe that changes each pixel's color using only that pixel: make it darker, flip it to its opposite, warm it up, or keep it only if it is "orange enough".

Formulation

As in Chapter 3, a point transformation is $g(x,y)=T[f(x,y)]$, but now f and g are color vectors. Written per component,

$$s_i=T_i(r_1,r_2,\dots,r_n),\qquad i=1,\dots,n,$$

where r_i and s_i are the i-th input and output components and n is the number of components ($n=3$ for RGB and HSI, $n=4$ for CMYK). Each output may depend on all inputs.

Any effect can be written in any model, but the cost differs. Scaling intensity by k is one multiplication in HSI ($s_3=k\,r_3$), three in RGB ($s_i=k\,r_i$), and an affine map in CMY ($s_i=k\,r_i+(1-k)$). Pick the model where the operation is natural, but remember that conversions cost time and amplify noise in dark or gray pixels, where hue and saturation are unstable.

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

Color complements

Hues opposite each other on the color circle are **complements**: red and cyan, green and magenta, blue and yellow. In RGB the complement is the photographic negative,

$$s_i=1-r_i,\qquad i=R,G,B,$$

which is the color version of the gray-level negative in Chapter 3. In HSI the hue complement is $H\mapsto(H+180^\circ)\bmod 360^\circ$, but the RGB negative also inverts intensity, so it cannot be computed from hue alone. Complements help reveal detail in dark regions.

Color slicing

Color slicing is intensity slicing for color: keep pixels whose color is close to a **prototype a** $= (a_R, a_G, a_B)$ and push the rest to a neutral color. Two common definitions of "close":

$$\text{cube:}\quad s_i = \begin{cases} 0.5, & \text{if } |r_j - a_j| > W/2 \text{ for any } j \\ r_i, & \text{otherwise} \end{cases} \qquad \text{sphere:}\quad s_i = \begin{cases} 0.5, & \text{if } \|\mathbf{r} - \mathbf{a}\| > R_0 \\ r_i, & \text{otherwise} \end{cases}$$

W is the width of a cube centered at **a**, R_0 the radius of a sphere, **r** the input color vector and 0.5 a mid-gray. The sphere treats all directions equally; the cube is cheaper and easy to reason about per channel. A friendlier variant keeps unmatched pixels as their own gray value instead of a flat gray, so context remains visible.

```
import numpy as np
from skimage import data, util, color

img = util.img_as_float(data.astronaut())
a = np.array([0.85, 0.40, 0.20])            # prototype: the orange of the suit
d = np.linalg.norm(img - a, axis=-1)        # distance of every pixel to a
gray = color.gray2rgb(color.rgb2gray(img))
sliced = np.where((d <= 0.25)[..., None], img, gray)
print((d <= 0.25).mean())                   # fraction kept, about 0.165
```

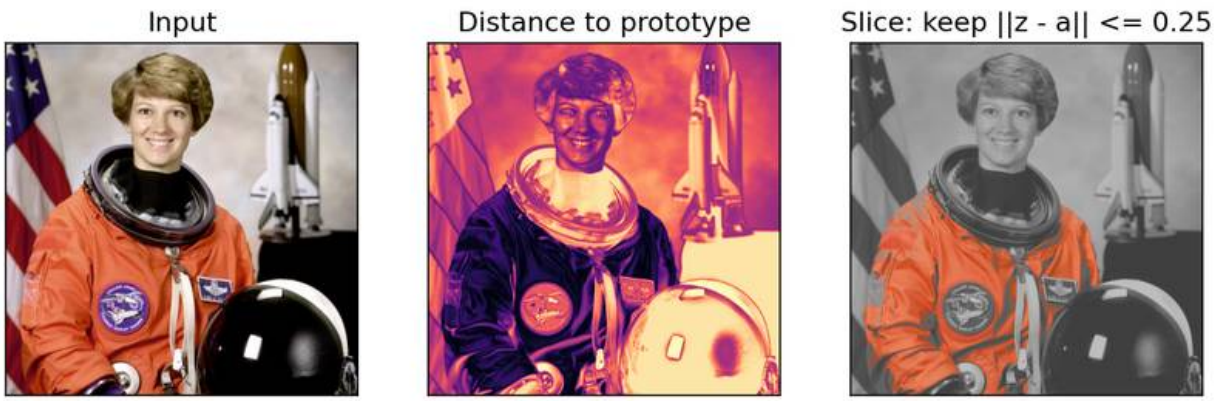


Figure 6.5 — Color slicing with a sphere of radius 0.25 around the suit's orange. Middle: distance to the prototype (dark = close). Right: matched pixels keep their color; everything else is shown in gray.

Tone and color corrections

Tone corrections adjust brightness and contrast and are best applied to an intensity-like channel (I, V or L^*) so hues are preserved. A mostly bright (high-key) image needs a curve that expands the highlights, a dark (low-key) image one that lifts the shadows, a flat image an S-curve.

Color balance corrections remove casts: to reduce a color, add its complement or reduce its neighbors on the color circle. Too much cyan? Increase red, or decrease blue and green. A gray card in the scene gives a precise target: after correction its R, G and B should be equal.

Real workflows convert each device's colors through a calibrated profile into a device-independent space such as L*a*b*, so a color looks the same on screen and on paper; colors outside a device's gamut (Figure 6.1) must be mapped inward.

White balance

White balance is the most common automatic color correction. A camera records surface color multiplied by the color of the light, so the same white wall looks orange under tungsten bulbs and bluish in shade. Our visual system largely discounts the light (**color constancy**); a camera has to do it computationally.

The standard model is diagonal: for illuminant color $\mathbf{e} = (e_R, e_G, e_B)$, the observed pixel is $c_i = e_i \rho_i$, where ρ_i is the surface's response under white light. Correction divides by the estimated illuminant, $\hat{\rho}_i = c_i / \hat{e}_i$ (usually rescaled to keep overall brightness). The hard part is estimating $\hat{\mathbf{e}}$ from one image. Classical estimators make an assumption about the scene:

- **Gray world** [10]: the average reflectance in a scene is achromatic, so $\hat{\mathbf{e}} \propto$ the mean RGB of the image.
- **White patch (max-RGB)** [11], [14]: the brightest surface is white, so $\hat{\mathbf{e}} \propto$ the per-channel maximum (in practice a high percentile, to ignore a few clipped pixels).
- **Gray edge** [11]: the average *edge* (image derivative) is achromatic. Van de Weijer, Gevers and Gijsenij show that all three are members of one family,

$$\left(\int \left| \frac{\partial^n \mathbf{c}_\sigma(\mathbf{x})}{\partial \mathbf{x}^n} \right|^p d\mathbf{x} \right)^{1/p} = k\, \hat{\mathbf{e}},$$

applied per channel, where $\mathbf{c}_\sigma$ is the image smoothed with a Gaussian of scale σ, n the derivative order, p a Minkowski norm and k a constant. $n = 0, p = 1$ gives gray world; $n = 0, p = \infty$ gives max-RGB; $n = 1$ gives gray-edge.

```
import numpy as np
from skimage import data, util

def white_balance(img, method="gray-world", p=99.5):
    """Diagonal white balance. img: float RGB in [0, 1]."""
    px = img.reshape(-1, 3)
    e = px.mean(0) if method == "gray-world" else np.percentile(px, p, axis=0)
    gain = e.mean() / e                 # one gain per channel
    return np.clip(img * gain, 0, 1), e

img = util.img_as_float(data.astronaut()) * 0.8
biased = np.clip(img * [0.75, 0.95, 1.25], 0, 1)      # add a bluish cast
fixed, e = white_balance(biased, "white-patch")
```

```
print(e.round(3))                                    # [0.598 0.757 0.996]
```

Drag the slider to compare the bluish input with the white-patch correction.



Bluish cast



White-patch corrected

Astronaut image with a synthetic blue cast and after white-patch white balance

Every one of these assumptions fails on some scene. On the cat photo below, whose true average color is orange-brown, gray world "corrects" the warm fur toward gray and leaves a blue tint, while white patch, helped by the white whiskers and highlights, does much better. The angular error between the estimated and true illuminant (the standard metric in this field) is printed above each result.



Figure 6.6 — Gray world versus white patch on an image whose average color is not gray. Angular errors come from our own synthetic experiment (see the figure script), not from a benchmark.

Histogram processing of color images

Equalizing the R, G and B histograms independently usually produces wrong colors, because it changes the *ratios* between channels and therefore the hue. The safer approach is to equalize only

an intensity channel and leave chromaticity alone:

1. Convert to a model that separates brightness (HSI, HSV, or L*a*b*).
2. Equalize I (or V, or L^*) using the Chapter 3 method.
3. Convert back.

```
import numpy as np
from skimage import data, util, exposure, color

img = util.img_as_float(data.astronaut())
per_channel = np.stack([exposure.equalize_hist(img[..., k]) for k in range(3)], -1)  # shifts
 ↪  hues
lab = color.rgb2lab(img)
lab[..., 0] = 100 * exposure.equalize_hist(lab[..., 0] / 100)                        #
 ↪  lightness only
better = np.clip(color.lab2rgb(lab), 0, 1)
```

Hue is preserved, but some colors may look washed out or leave the gamut on the way back, so a clip or mild saturation boost is sometimes needed. Local histogram methods (Chapter 3) apply to L^* the same way.

Color image smoothing and sharpening

In plain words. To blur or sharpen a color photo you can blur or sharpen each of the three channels, because these are linear operations. But doing it on brightness only is often cleaner.

Smoothing

Neighborhood averaging of a color image is

$$\bar{\mathbf{c}}(x,y) = \frac{1}{K} \sum_{(s,t)\in S_{xy}} \mathbf{c}(s,t),$$

where S_{xy} is the neighborhood and K its size. As shown earlier, this equals averaging each channel separately, so `cv2.GaussianBlur` on a 3-channel array does the right thing.

An alternative is to smooth only the intensity of an HSI (or L^*) image and keep hue and saturation. The results differ near edges: per-channel smoothing creates intermediate colors along a red–green boundary (an olive line), while intensity-only smoothing keeps the original hues and only softens brightness. Which you want depends on the task: denoising usually prefers smoothing chroma *more* than luma, since the eye is less sensitive to chroma blur (the same fact that YCbCr subsampling exploits).

Sharpening

The Laplacian of a vector is the vector of the Laplacians of its components:

$$\nabla^2\mathbf{c}(x,y) = \begin{bmatrix} \nabla^2 R(x,y) \\ \nabla^2 G(x,y) \\ \nabla^2 B(x,y) \end{bmatrix},$$

so Laplacian sharpening, $\mathbf{g} = \mathbf{c} - k\,\nabla^2\mathbf{c}$ with strength $k > 0$, can be done per channel. Unsharp masking (Chapter 3) works the same way. In practice sharpening only the intensity channel avoids colored halos ("fringes") around high-contrast edges, which appear when the three channels overshoot by different amounts.

Using color in image segmentation

In plain words. Color is one of the easiest clues for splitting an image into parts: "everything brown is stained tissue, everything blue is background." The question is how to measure "brown enough".

Segmentation in HSI space

If the target is described by its hue, HSI is a natural choice. A typical recipe: threshold saturation to build a mask of pixels that actually have a color (hue is meaningless at low saturation), multiply the hue image by this mask, then threshold or histogram the masked hue to pick the target range. The weakness is the hue singularity: noise in near-gray pixels produces random hues, which is why the saturation mask comes first.

Segmentation in RGB vector space

RGB vectors often give better results because they avoid the unstable hue computation. Choose a set of sample pixels from the target, compute their mean color $\mathbf{a}$, and classify every pixel $\mathbf{z}$ by its distance to $\mathbf{a}$. The **Euclidean distance** is

$$D(\mathbf{z},\mathbf{a}) = \|\mathbf{z}-\mathbf{a}\| = \left[(\mathbf{z}-\mathbf{a})^{\mathsf{T}}(\mathbf{z}-\mathbf{a})\right]^{1/2},$$

and pixels with $D \le D_0$ are labeled as target. The set $D \le D_0$ is a solid sphere of radius D_0. Sample colors, however, usually spread more in some directions than others (for stained tissue, mostly along the light-to-dark direction). The **Mahalanobis distance** accounts for that:

$$D(\mathbf{z},\mathbf{a}) = \left[(\mathbf{z}-\mathbf{a})^{\mathsf{T}}\mathbf{C}^{-1}(\mathbf{z}-\mathbf{a})\right]^{1/2},$$

where $\mathbf{C}$ is the 3×3 covariance matrix of the samples. Now $D \le D_0$ is an ellipsoid aligned with the data. A cheaper approximation is a **bounding box** centered at $\mathbf{a}$ whose half-width along each axis is a multiple of that channel's standard deviation; testing a box needs only comparisons, no squares or square roots.

```
import numpy as np
from skimage import data, util

img = util.img_as_float(data.immunohistochemistry())
samples = img[60:100, 60:100].reshape(-1, 3)        # a brown (stained) region
a, C = samples.mean(0), np.cov(samples.T)
z = img.reshape(-1, 3) - a
d_euc = np.sqrt((z ** 2).sum(1)).reshape(img.shape[:2])
d_mah = np.sqrt(np.einsum("ij,jk,ik->i", z, np.linalg.inv(C), z)).reshape(img.shape[:2])
mask_euc, mask_mah = d_euc <= 0.25, d_mah <= 3.0
```

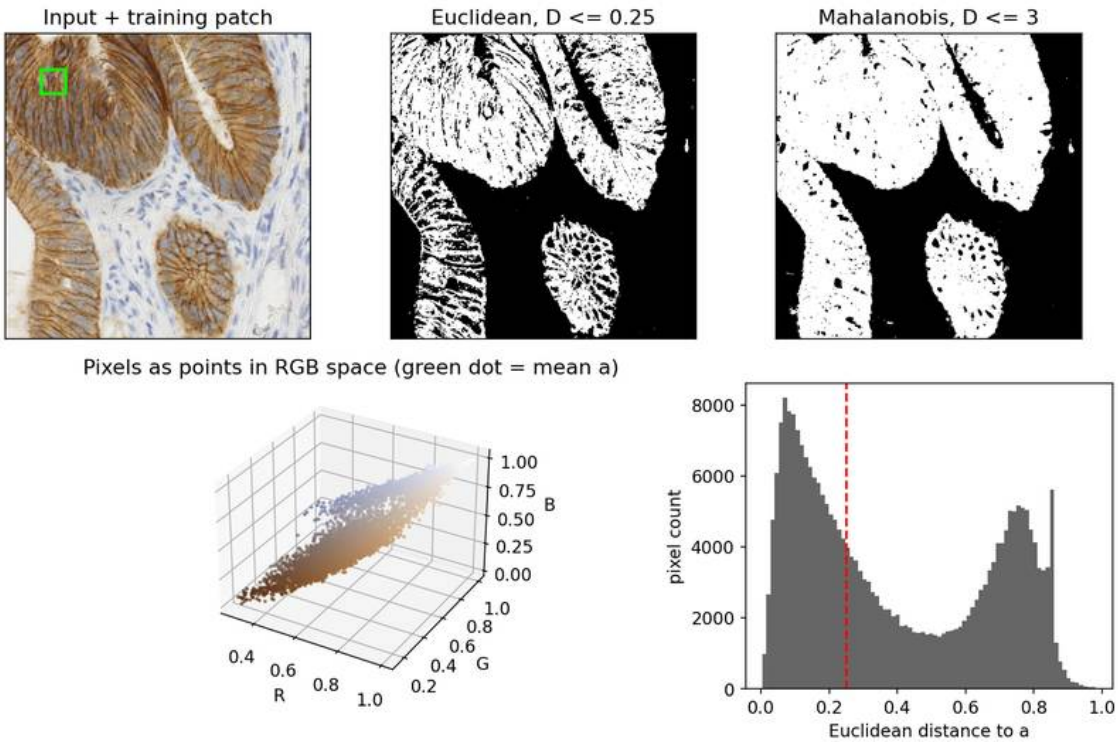


Figure 6.7 — Segmentation in RGB vector space. Top: stained tissue sample with the training patch (green box); Euclidean sphere with radius 0.25; Mahalanobis ellipsoid with radius 3. The ellipsoid follows the elongated brown cluster and fills the stained regions more completely. Bottom: pixels form an elongated cloud from dark brown to pale blue; the distance histogram is bimodal, which guides the choice of threshold.

Color edge detection

Computing a gradient on each channel and adding the results is a common shortcut, but it is not the gradient of the vector image. Worse, edge detection on a grayscale version misses **isoluminant** edges, boundaries between two colors of the same brightness. Di Zenzo [13] defined a proper gradient for multi-channel images. With $\mathbf{u} = (\partial R/\partial x, \partial G/\partial x, \partial B/\partial x)$ and $\mathbf{v} = (\partial R/\partial y, \partial G/\partial y, \partial B/\partial y)$, define

$$g_{xx} = \mathbf{u} \cdot \mathbf{u}, \qquad g_{yy} = \mathbf{v} \cdot \mathbf{v}, \qquad g_{xy} = \mathbf{u} \cdot \mathbf{v}.$$

These form the 2×2 structure tensor $\begin{bmatrix} g_{xx} & g_{xy} \\ g_{xy} & g_{yy} \end{bmatrix}$, which summarizes how fast the color vector changes in every direction. The edge strength is the square root of its largest eigenvalue,

$$\lambda_{\max} = \tfrac{1}{2}\Big[g_{xx} + g_{yy} + \sqrt{(g_{xx} - g_{yy})^2 + 4g_{xy}^2}\Big],$$

so $\sqrt{\lambda_{\max}}$ is the largest rate of change of the color vector over all directions. Unlike a gradient of the gray image, it responds whenever *any* channel changes, so it sees isoluminant edges.

Online only: Di Zenzo's gradient direction — https://blog.shyandram.dev/en/learn/image-processing/color/notes/di-zenzos-gradient-direction/

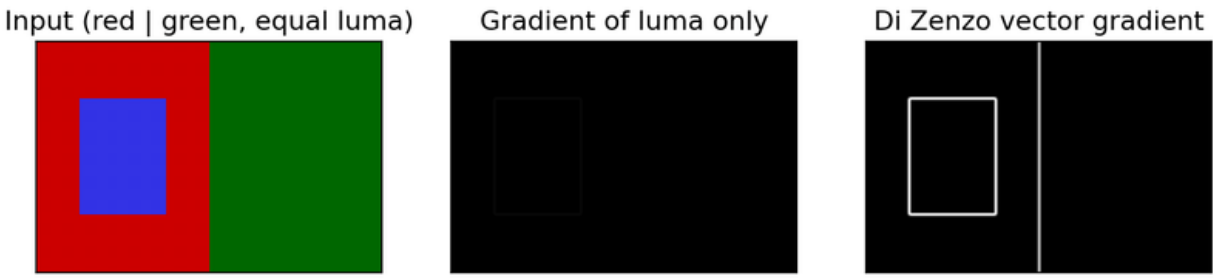


Figure 6.8 — An isoluminant edge. The red and green halves have the same BT.601 luma, so a gradient of the luma image sees no boundary between them (and only a faint square). The Di Zenzo vector gradient finds both.

```
import numpy as np
from skimage import filters

def di_zenzo(img):
    """Edge strength of a float RGB image via the largest structure-tensor eigenvalue."""
    gx = np.stack([filters.sobel_v(img[..., k]) for k in range(3)], -1)   # d/dx
    gy = np.stack([filters.sobel_h(img[..., k]) for k in range(3)], -1)   # d/dy
    gxx, gyy, gxy = (gx * gx).sum(-1), (gy * gy).sum(-1), (gx * gy).sum(-1)
    return np.sqrt(0.5 * (gxx + gyy + np.sqrt((gxx - gyy) ** 2 + 4 * gxy ** 2)))
```

Noise in color images

In plain words. Each color channel gets its own noise. When you turn the noisy channels back into hue and saturation, the noise mixes and can look much worse, especially in dark or gray areas.

The noise models of Chapter 5 apply to each channel. In a camera, noise arises mostly in the sensor, before color is reconstructed. Most sensors sit behind a color filter array, so each photosite measures one color and the other two are interpolated (**demosaicking**); the interpolation spreads and correlates noise across channels. Lukac et al. [12] review this whole chain.

Two observations shape practice:

- **Color conversions redistribute noise.** Intensity $I = (R + G + B)/3$ averages three channels, so independent zero-mean noise with variance σ^2 per channel becomes $\sigma^2/3$ in I: intensity is *cleaner* than any channel. Hue and saturation, in contrast, are nonlinear functions with large derivatives near the gray axis, so they become much *noisier*. This is why denoisers often work in a luma–chroma space and smooth chroma more strongly.
- **Impulse noise in one channel is a colored speck.** A "salt" spike in the green channel alone makes a bright green dot. A per-channel median removes it, but in mixed cases can create colors that were not in the neighborhood.

The **vector median filter** avoids invented colors. In each window it outputs the input vector that minimizes the sum of distances to all other vectors in the window [12]:

$$\mathbf{c}_{\mathrm{VM}} = \arg\min_{\mathbf{c}_i \in W} \sum_{\mathbf{c}_j \in W} \|\mathbf{c}_i - \mathbf{c}_j\|,$$

where W is the set of color vectors in the window. The output is always one of the inputs, and for one-dimensional data it reduces to the ordinary median.

```
import numpy as np
from skimage import data, util

def vector_median(img, k=3):
    r = k // 2
    H, W, _ = img.shape
    p = np.pad(img, ((r, r), (r, r), (0, 0)), mode="edge")
    nb = np.stack([p[i:i + H, j:j + W] for i in range(k) for j in range(k)], 2)  # (H, W,
     ↪  k*k, 3)
    D = np.linalg.norm(nb[:, :, :, None] - nb[:, :, None, :], axis=-1).sum(-1)    # (H, W,
     ↪  k*k)
    return np.take_along_axis(nb, D.argmin(-1)[..., None, None], 2)[:, :, 0]

img = util.img_as_float(data.astronaut())
rng = np.random.default_rng(0)
noisy = img.copy()
hit = rng.random(img.shape[:2]) < 0.05
noisy[hit] = rng.random((hit.sum(), 3))          # 5% random color impulses
out = vector_median(noisy)
psnr = lambda x: 10 * np.log10(1 / ((x - img) ** 2).mean())
print(round(psnr(noisy), 1), round(psnr(out), 1))  # about 20.4 -> 31.3 dB
```

This vectorized form uses memory proportional to k^4 per pixel; use a loop for large windows.

Color image compression

In plain words. A color image has three times the data of a gray one, but much is redundant: the channels look alike, and our eyes see little color detail. Codecs exploit both facts:

1. **Decorrelate the channels.** R, G and B are strongly correlated (Figure 6.2 shows three very similar images). Converting to luma plus two chroma differences, such as YCbCr [6], concentrates most of the energy in one channel.
2. **Subsample chroma.** Because the eye resolves less color detail than brightness detail, the chroma channels can be stored at half resolution horizontally (4:2:2) or in both directions (4:2:0) with little visible loss. 4:2:0 alone halves the raw data: $1 + 2 \cdot \frac{1}{4} = 1.5$ samples per pixel instead of 3.

Each channel is then compressed with the transform, quantization and entropy coding methods of Chapter 8. Typical color artifacts are bleeding across sharp edges (from chroma subsampling) and blocky color patches at high compression.

Modern view

The classical tools above still run inside every camera and image library. What changed after about 2010 is mostly *how the parameters are chosen*: learned from data instead of fixed by assumption. Five reviews and survey-style papers map the territory.

Online only: The Modern view in more detail — https://blog.shyandram.dev/en/learn/image-processing/color/papers/the-modern-view-in-more-detail/

Color constancy: from assumptions to learning. Gijsenij, Gevers and van de Weijer [14] survey computational color constancy and sort methods into three groups: fixed assumptions (gray world, white patch, gray edge), gamut-based methods, and methods learned from images with known illuminants. A recurring finding is that no single fixed assumption suits every scene. Learned estimators followed: Fast Fourier Color Constancy (FFCC) [15] runs in real time on phones, FC4 [16] lets every image patch vote for the illuminant with a learned confidence, and Afifi and Brown [17] correct white balance even after the camera has rendered a nonlinear sRGB image. The diagonal model is still the backbone; the estimate is now learned.

Vector filtering for color. Lukac et al. [12] review filters that treat color pixels as vectors. Their takeaway matches this chapter: componentwise processing can create color artifacts, while vector order statistics keep the input's colors. Learned denoisers are now stronger when training data and compute are available, but these filters remain standard baselines where speed and predictability matter.

Colormap design. Kovesi [7], the viridis family [8] and Crameri et al. [9] share one principle: perceived lightness must change uniformly along the map, and rainbow and red–green maps should leave software defaults. Deep learning changed nothing here: the problem is human perception, and the answer is careful design measured in a perceptual color space.

Colorization with deep learning. Pseudocolor maps gray to color by a fixed rule; *colorization* tries to predict the *true* colors. Zhang, Isola and Efros [18] treat it as classification over quantized colors, Anwar et al. [19] survey the field, and DDColor [20] (ICCV 2023) reduces color bleeding across object boundaries. Two classical ideas persist: work in L*a*b* (predict chroma, keep the given lightness) and treat color as a distribution, because a gray shirt could have been any color.

What did not change. Colorimetry (CIE XYZ, chromaticity, gamuts), encodings (sRGB, BT.601/709 YCbCr) and color-difference spaces are standards, not things to learn. Networks are trained on sRGB data and judged in these spaces. Knowing exactly which space your tensors are in (encoded or linear, BT.601 or BT.709, studio or full range) remains one of the most practical skills in color work.

Key takeaways

- Human vision is trichromatic, so three numbers describe a color for a human observer; different spectra with equal responses (metamers) look the same.
- The CIE xy chromaticity diagram shows all visible chromaticities; a three-primary device covers only a triangle (its gamut).
- RGB suits hardware, CMY(K) suits ink, HSI separates intensity from hue and saturation, and L*a*b* makes Euclidean distance roughly perceptual. Watch for undefined hue on grays, sRGB gamma, and BT.601 versus BT.709.
- Pseudocolor makes gray data readable; use perceptually uniform colormaps such as viridis, not rainbow maps.
- Linear operations (smoothing, Laplacian sharpening) can be done per channel; nonlinear ones (median, histogram equalization, edge detection, segmentation) should treat pixels as vectors or work on an intensity channel.
- White balance divides by an estimated illuminant; gray world, white patch and gray edge are special cases of one framework, and modern systems learn the estimate.
- Segment color with vector distances (Euclidean, Mahalanobis, box), and detect color edges with the Di Zenzo gradient, which sees isoluminant boundaries.
- Intensity is less noisy than any single channel, hue and saturation are more noisy; vector median filters remove color impulses without inventing new colors.

Exercises

1. A pixel has $(R, G, B) = (0.6, 0.6, 0.2)$. Compute its H, S and I by hand. Then predict, without computing, how H changes if B increases to 0.3.

   Hint

   $I = 0.4667$. min $= 0.2$, so $S = 1 - 0.2/0.4667 \approx 0.571$. For θ: numerator $= 0.5(0 + 0.4) = 0.2$, denominator $= \sqrt{0 + 0.4 \cdot 0.4} = 0.4$, so $\theta = \cos^{-1}(0.5) = 60°$, and since $B \leq G$, $H = 60°$ (yellow). Raising B to 0.3 keeps $R = G$, so hue stays 60°; only S drops.

2. Show that the RGB negative $s_i = 1 - r_i$ maps a pixel with HSI coordinates (H, S, I) to one whose intensity is $1 - I$ and whose hue is $H + 180°$ (for non-gray pixels). Is the saturation preserved?

   Hint

Intensity follows from linearity. For hue, the chromatic part $\mathbf{c} - I\,\mathbf{1}$ simply changes sign, which rotates it by 180° around the gray axis. Saturation is $1 - \min / I$; after negation the minimum becomes $1 - \max(R, G, B)$ and the intensity $1 - I$, so in general saturation changes. Try $(0.9, 0.1, 0.1)$.

3. You equalize the R, G and B histograms of a photo of a blue sky separately. Explain why the sky may turn gray-cyan, and propose two fixes.

Hint

Equalization stretches each channel to use the full range. A channel that was almost uniformly high (blue) can be pushed down relative to the others, while weak channels are boosted, so the R:G:B ratio changes. Fix by equalizing I, V or L^* only, or by applying one common mapping computed from the average histogram to all three channels.

4. In the gray-world/white-patch framework, what estimator do you get with $n = 0$ and $p = 2$? Run the `white_balance` idea on the cat image with $p = 1, 2, 6, \infty$ (use a high percentile for ∞) and plot the angular error versus p.

Hint

$p = 2$ estimates the illuminant from the root-mean-square value of each channel, a compromise between the mean and the maximum. With the chapter's synthetic cast the error falls from about 12° ($p = 1$) to about 11° ($p = 2$), 7° ($p = 6$) and 4° ($p = 20$). It falls as p grows on this image, because its bright white details are a better clue than its orange average. On a scene without white objects the trend can reverse.

5. Design a color segmenter for green vegetation in outdoor photos that is robust to shadows. Which space and which distance would you use, and why might plain RGB Euclidean distance fail?

Hint

Shadows scale all three channels down, which moves a pixel along a line through the origin and changes its RGB Euclidean distance a lot. Use a chromaticity-based feature that is unaffected by scaling (normalized $r = R/(R+G+B)$, $g = G/(R+G+B)$, or hue with a saturation mask, or a^*, b^*), or a Mahalanobis distance whose covariance is trained on sunlit and shaded samples.

6. Explain why the vector median filter never creates a new color, and give a 3-pixel, 1-D example where the per-channel median does.

Hint

By definition the vector median is one of the window's input vectors. For three pixels $(1, 1, 0)$, $(1, 0, 1)$, $(0, 1, 1)$ (yellow, magenta, cyan) the per-channel median is $(1, 1, 1)$, pure white, which appears nowhere in the window.

The figures in this chapter are generated by `scripts/figures/dip_ch06.py` from `skimage.data` images and synthetic arrays.

References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018, Ch. 6. publisher page
2. C. Wyman, P.-P. Sloan, and P. Shirley, "Simple Analytic Approximations to the CIE XYZ Color Matching Functions," *Journal of Computer Graphics Techniques*, vol. 2, no. 2, 2013. jcgt.org
3. IEC 61966-2-1:1999, "Multimedia systems and equipment – Colour measurement and management – Part 2-1: Colour management – Default RGB colour space – sRGB," IEC, 1999. IEC webstore
4. Recommendation ITU-R BT.709-6, "Parameter values for the HDTV standards for production and international programme exchange," ITU, 2015. ITU
5. ISO/CIE 11664-4:2019, "Colorimetry – Part 4: CIE 1976 L*a*b* colour space," ISO/CIE, 2019. standard page
6. Recommendation ITU-R BT.601-7, "Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios," ITU, 2011. ITU
7. P. Kovesi, "Good Colour Maps: How to Design Them," arXiv:1509.03700, 2015. arXiv
8. S. van der Walt and N. Smith, viridis colormap design notes (BIDS project page; presented at SciPy 2015), 2015. project page
9. F. Crameri, G. E. Shephard, and P. J. Heron, "The misuse of colour in science communication," *Nature Communications*, vol. 11, art. 5444, 2020. PMC
10. G. Buchsbaum, "A spatial processor model for object colour perception," *Journal of the Franklin Institute*, vol. 310, no. 1, pp. 1–26, 1980. DOI
11. J. van de Weijer, T. Gevers, and A. Gijsenij, "Edge-Based Color Constancy," *IEEE Transactions on Image Processing*, vol. 16, no. 9, pp. 2207–2214, 2007. DOI
12. R. Lukac, B. Smolka, K. Martin, K. N. Plataniotis, and A. N. Venetsanopoulos, "Vector Filtering for Color Imaging," *IEEE Signal Processing Magazine*, vol. 22, no. 1, pp. 74–86, Jan. 2005. PDF
13. S. Di Zenzo, "A note on the gradient of a multi-image," *Computer Vision, Graphics, and Image Processing*, vol. 33, pp. 116–125, 1986. DOI
14. A. Gijsenij, T. Gevers, and J. van de Weijer, "Computational Color Constancy: Survey and Experiments," *IEEE Transactions on Image Processing*, vol. 20, no. 9, pp. 2475–2489, 2011. DOI
15. J. T. Barron and Y.-T. Tsai, "Fast Fourier Color Constancy," *CVPR*, 2017. arXiv
16. Y. Hu, B. Wang, and S. Lin, "FC4: Fully Convolutional Color Constancy with Confidence-Weighted Pooling," *CVPR*, 2017. paper
17. M. Afifi and M. S. Brown, "Deep White-Balance Editing," *CVPR*, 2020. arXiv
18. R. Zhang, P. Isola, and A. A. Efros, "Colorful Image Colorization," arXiv:1603.08511, 2016. arXiv
19. S. Anwar, M. Tahir, C. Li, A. Mian, F. S. Khan, and A. W. Muzaffar, "Image Colorization: A Survey and Dataset," arXiv:2008.10774, 2020. arXiv
20. X. Kang, T. Yang, W. Ouyang, P. Ren, L. Li, and X. Xie, "DDColor: Towards Photo-Realistic Image Colorization via Dual Decoders," *ICCV*, 2023. arXiv

IP 07 • Wavelet and Other Image Transforms

Treat every image transform as a change of basis, then build up from the DCT and Walsh–Hadamard transforms to Haar, multiresolution analysis and the fast wavelet transform.

Online edition: https://blog.shyandram.dev/en/learn/image-processing/wavelets-transforms/

What you'll learn

- Why every linear image transform (Fourier, cosine, Walsh–Hadamard, Haar, wavelet) is the same idea: describe the image in a different set of building blocks, called a **basis**.
- How to write a transform as a pair of matrix products, and why separable 2-D transforms are cheap.
- What basis images look like for the DCT, Walsh–Hadamard, slant and Haar transforms, and why the DCT is the workhorse of JPEG.
- How the time–frequency plane explains the difference between Fourier analysis and wavelet analysis.
- How multiresolution analysis, scaling functions and wavelet functions lead to the fast wavelet transform, a simple two-filter bank.
- How to use the 2-D discrete wavelet transform for denoising, edge detection and compression, with PyWavelets.

The big picture

In Chapter 4 you described an image as a sum of sinusoids. That is one choice of building blocks, but not the only one. This chapter steps back and asks a more general question: *given any set of building blocks, how do I find out how much of each one is in my image, and which set is best for the job?*

> **Explain like I'm five**
>
> Imagine describing a LEGO model to a friend over the phone. You could list every single stud ("red at row 1, column 1; red at row 1, column 2; ..."). That always works, but it is slow. Or you could say "a 2×4 red brick here, a 1×2 blue brick there". Now the description is short, because the bricks match the shapes in the model.
>
> An image transform is a choice of bricks. The pixel grid uses tiny 1×1 bricks. The Fourier and cosine transforms use stripy bricks. Wavelets use bricks of every size: big blurry ones for the sky, tiny sharp ones for edges. When the bricks match the picture, only a few of them are needed, and that is what makes compression, denoising and fast analysis possible.

The rigorous version of "choose your bricks" is linear algebra: inner products, bases and change of basis. Once that machinery is in place, every transform in this chapter is just a different matrix.

Wavelets get the most attention, because they are the first basis that is localized in *both* space and frequency. That property made them the core of the JPEG 2000 standard [15] and an important idea behind modern sparse modeling and, more recently, some neural-network layers.

Preliminaries

Plain version: a signal with N samples is an arrow (a vector) in an N-dimensional space. A transform measures the shadow that arrow casts on each of N reference arrows. If the reference arrows are perpendicular and of unit length, the shadows tell you everything and you can rebuild the arrow by adding the shadows back.

Inner product spaces

An **inner product** $\langle \mathbf{f}, \mathbf{g} \rangle$ takes two vectors and returns a number that measures how much they point the same way. For real or complex vectors of length N,

$$\langle \mathbf{f}, \mathbf{g} \rangle = \sum_{x=0}^{N-1} f(x)\, g^{*}(x), \qquad \|\mathbf{f}\| = \sqrt{\langle \mathbf{f}, \mathbf{f} \rangle}.$$

Here $f(x)$ and $g(x)$ are the x-th entries of $\mathbf{f}$ and $\mathbf{g}$, g^{*} is the complex conjugate (it does nothing for real data), and $\|\mathbf{f}\|$ is the length, or **norm**, of $\mathbf{f}$. For continuous functions, the sum becomes an integral, $\langle f, g \rangle = \int f(x) g^{*}(x)\, dx$. Two vectors are **orthogonal** when their inner product is zero: they share nothing.

Orthonormal bases

A set of vectors $\{\mathbf{s}_0, \dots, \mathbf{s}_{N-1}\}$ is an **orthonormal basis** if every vector has unit length and every pair is orthogonal: $\langle \mathbf{s}_u, \mathbf{s}_v \rangle = \delta(u - v)$, where δ is 1 when $u = v$ and 0 otherwise. Then any vector splits cleanly into

$$t(u) = \langle \mathbf{f}, \mathbf{s}_u \rangle, \qquad \mathbf{f} = \sum_{u=0}^{N-1} t(u)\, \mathbf{s}_u.$$

The numbers $t(u)$ are the **transform coefficients**. The first equation is the **analysis** (forward) step and the second is the **synthesis** (inverse) step. Orthonormal bases also preserve energy, $\sum_x |f(x)|^2 = \sum_u |t(u)|^2$ (Parseval's theorem), which is why we can talk about "how much energy" sits in each coefficient.

Biorthogonal bases

Sometimes the most useful building blocks are not perpendicular. Then you need two families: an analysis set $\{\tilde{\mathbf{s}}_u\}$ for measuring and a synthesis set $\{\mathbf{s}_u\}$ for rebuilding. They form a **biorthogonal** pair when

$$\langle \mathbf{s}_u, \tilde{\mathbf{s}}_v \rangle = \delta(u - v), \qquad t(u) = \langle \mathbf{f}, \tilde{\mathbf{s}}_u \rangle, \qquad \mathbf{f} = \sum_u t(u)\, \mathbf{s}_u.$$

The "measuring sticks" $\tilde{\mathbf{s}}_u$ are the **dual** basis. Biorthogonality buys freedom. For example, JPEG 2000 uses biorthogonal wavelet filters, which can be short *and* symmetric, a combination that is hard to get with orthonormal wavelets [15].

Frames

A **frame** is a spanning set with *more* vectors than dimensions, so it is redundant. Its defining property is that the measured energy stays within fixed bounds:

$$A\,\|\mathbf{f}\|^2 \;\le\; \sum_k |\langle \mathbf{f}, \mathbf{s}_k \rangle|^2 \;\le\; B\,\|\mathbf{f}\|^2, \qquad 0 < A \le B < \infty.$$

A and B are the **frame bounds**. If $A = B$ the frame is **tight**, and reconstruction is just $\mathbf{f} = \frac{1}{A}\sum_k \langle \mathbf{f}, \mathbf{s}_k \rangle \mathbf{s}_k$. A classic picture: three unit arrows in the plane, 120° apart. They span the plane, no two are orthogonal, and they form a tight frame with $A = B = 3/2$. Redundant frames are worth the extra coefficients when you want robustness or shift invariance, as in the undecimated wavelet transform and the scattering transform discussed in the Modern view.

```
import numpy as np

v = np.array([3.0, 1.0, 2.0])
e = np.eye(3)                          # standard orthonormal basis
c = e @ v                              # coefficients = inner products <v, e_k>
print(c, np.allclose(c @ e, v))        # synthesis: sum_k c_k e_k

# A biorthogonal pair in 2-D: analysis rows A, synthesis columns B = A^{-1}
A = np.array([[1.0, 0.0], [1.0, 1.0]])
B = np.linalg.inv(A)
x = np.array([2.0, 5.0])
print(A @ x, B @ (A @ x))              # [2. 7.] -> back to [2. 5.]
```

Matrix-based transforms

Plain version: stack the measuring vectors as the rows of a matrix. Multiplying the signal by that matrix measures all the coefficients at once; multiplying by the inverse rebuilds the signal.

For a 1-D signal $\mathbf{f}$ of length N, put the basis vectors $\mathbf{s}_u^{*T}$ in the rows of an $N \times N$ **transformation matrix** $\mathbf{A}$. Then

$$\mathbf{t} = \mathbf{A}\,\mathbf{f}, \qquad \mathbf{f} = \mathbf{A}^{-1}\mathbf{t}.$$

When the basis is orthonormal, $\mathbf{A}$ is **unitary** ($\mathbf{A}^{-1} = \mathbf{A}^{*T}$; for real matrices, **orthogonal**, $\mathbf{A}^{-1} = \mathbf{A}^T$), so the inverse is free.

For an $M \times N$ image $\mathbf{F}$, a general 2-D linear transform would need an $MN \times MN$ matrix. Nearly all practical image transforms are **separable**: they apply a 1-D transform down every column and then along every row. In matrix form,

$$\mathbf{T} = \mathbf{A}\,\mathbf{F}\,\mathbf{B}^T, \qquad \mathbf{F} = \mathbf{A}^{-1}\,\mathbf{T}\,(\mathbf{B}^T)^{-1},$$

where $\mathbf{A}$ $(M \times M)$ acts on columns and $\mathbf{B}$ $(N \times N)$ on rows; for a square image and the same transform in both directions, $\mathbf{B} = \mathbf{A}$. Separability cuts the cost of a direct $N \times N$ transform from $O(N^4)$ to $O(N^3)$ operations, and fast algorithms (FFT, fast DCT, fast wavelet transform) reduce it further.

```
import numpy as np
from scipy.fft import dct, dctn
from skimage import data, img_as_float

N = 8
A = dct(np.eye(N), norm="ortho", axis=0)  # row u = u-th cosine basis vector

print(np.allclose(A @ A.T, np.eye(N)))    # orthonormal -> True

F = img_as_float(data.camera())[200:208, 200:208]
T = A @ F @ A.T                           # forward 2-D transform
print(np.allclose(T, dctn(F, norm="ortho")))   # same as scipy -> True
print(np.allclose(A.T @ T @ A, F))             # inverse -> True
```

Correlation

Plain version: each coefficient answers the question "how much does my image look like this particular pattern?" That question is a correlation.

Compare the analysis equation with the cross-correlation from Chapter 3. The coefficient

$$t(u) = \langle \mathbf{f}, \mathbf{s}_u \rangle = \sum_x f(x)\, s_u^*(x)$$

is exactly the correlation of f with the basis function s_u, evaluated at zero shift. A large $|t(u)|$ means f and s_u are similar; $t(u) = 0$ means f contains none of s_u. Two consequences follow:

- **Transforms are template matching.** Each basis function is a template, and the transform scores every template at once. The Fourier transform scores sinusoids; the Haar transform scores step-like patterns.
- **Wavelets are correlation at many shifts and scales.** The wavelet basis contains shifted and stretched copies of a single pattern. Correlating with all the shifts of one copy is a filtering operation, which is why the wavelet transform will turn into a bank of filters later in this chapter.

Basis functions in the time-frequency plane

Plain version: some building blocks tell you *when* something happened, some tell you *which pitch* it had, and none can tell you both perfectly. The time–frequency plane is a map of that trade-off.

For a basis function $s(x)$ with unit energy, define its spread in time (or space) and in frequency as standard deviations of $|s(x)|^2$ and $|S(\omega)|^2$:

$$\sigma_x^2 = \int (x - \bar{x})^2 |s(x)|^2 \, dx, \qquad \sigma_\omega^2 = \frac{1}{2\pi} \int (\omega - \bar{\omega})^2 |S(\omega)|^2 \, d\omega,$$

where $S(\omega)$ is the Fourier transform of s, and $\bar{x}$, $\bar{\omega}$ are the centers of mass of the two energy distributions. The **Heisenberg uncertainty principle** for signals states

$$\sigma_x \, \sigma_\omega \;\geq\; \tfrac{1}{2},$$

with equality only for Gaussian-shaped functions. Draw each basis function as a rectangle (a **Heisenberg box** or tile) of width σ_x and height σ_ω centered at $(\bar{x}, \bar{\omega})$. The area can never shrink below a fixed minimum, but the *shape* is up to you.

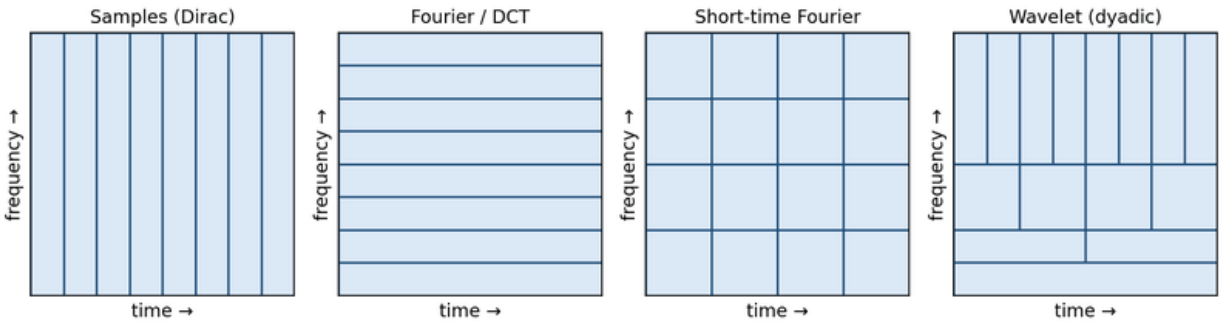


Figure 7.1 — How four bases tile the time–frequency plane. Every tile has the same area. Samples are sharp in time; Fourier/DCT bases are sharp in frequency; the short-time Fourier transform fixes one compromise for all frequencies; wavelets adapt the shape to the frequency.

Figure 7.1 shows the four standard cases:

- **Samples (the identity basis):** perfect position, no frequency information.
- **Fourier or DCT:** perfect frequency, no position. A single sharp edge spreads over all coefficients.
- **Short-time Fourier transform (STFT):** cut the signal into windows and take a Fourier transform of each. Every tile has the same shape, chosen once for all frequencies [2].
- **Wavelets:** high-frequency tiles are narrow in time and tall in frequency; low-frequency tiles are wide and short. This matches natural images well: edges are brief, high-frequency events, while smooth shading is a slow, low-frequency one.

Basis images

Plain version: for a 2-D transform, each coefficient has its own little picture. The image is a weighted sum of those pictures, and the coefficient is the weight.

For a separable transform with matrix $\mathbf{A}$ whose u-th row is $\mathbf{a}_u^T$, rewrite the inverse as a sum of outer products:

$$\mathbf{F} = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} T(u,v)\, \mathbf{S}_{u,v}, \qquad \mathbf{S}_{u,v} = \mathbf{a}_u^* \mathbf{a}_v^{*T}.$$

Each $N \times N$ matrix $\mathbf{S}_{u,v}$ is a **basis image**. $T(u,v)$ tells you how much of basis image (u,v) to add, u indexes the vertical pattern and v the horizontal one. Plotting all N^2 basis images side by side is the fastest way to understand what a transform "sees".

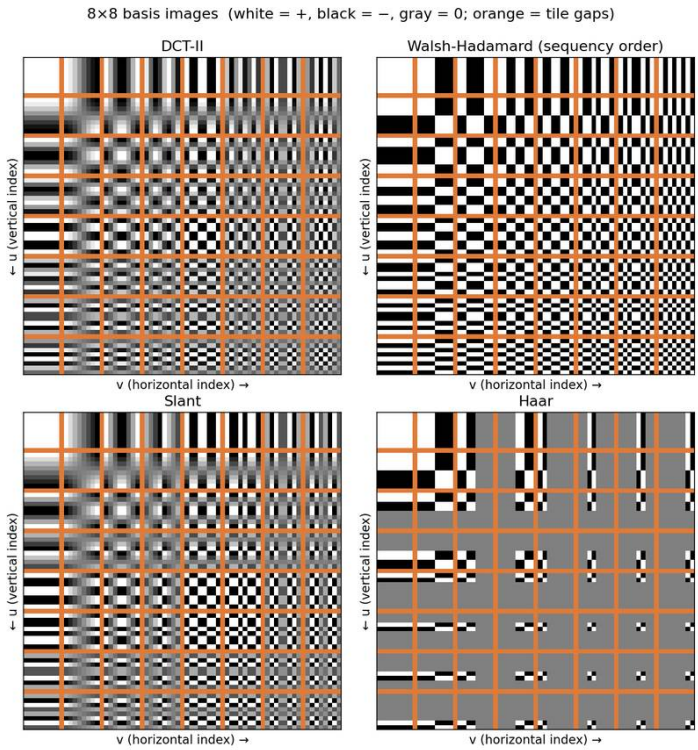


Figure 7.2 — 8×8 basis images of four transforms, arranged so that vertical frequency (sequency) grows downward and horizontal frequency grows to the right. Each tile is scaled to $[-1, 1]$. Note how Haar basis images, unlike the other three, are zero (gray) over most of the tile: they are localized.

The top-left basis image is constant in all four transforms: it measures the average brightness, called the **DC** coefficient. Moving right or down, the patterns oscillate faster. The DCT oscillates smoothly, Walsh–Hadamard switches abruptly between ± 1, slant adds linear ramps, and Haar confines each pattern to a small region of the tile.

Fourier-related transforms

Plain version: the Fourier family uses waves as building blocks. The cosine transform is the member that behaves best at the borders of a block, which is why your phone's JPEG files use it.

The discrete Fourier transform, as a matrix

The 1-D DFT from Chapter 4 is a matrix transform with

$$A_{u,x} = \frac{1}{\sqrt{N}} e^{-j2\pi ux/N}, \qquad u, x = 0, \dots, N-1.$$

$A_{u,x}$ is the entry in row u, column x; $j = \sqrt{-1}$. With the $1/\sqrt{N}$ factor the matrix is unitary. Two drawbacks for compression: the coefficients are complex, and the DFT silently treats the signal as periodic. If the left and right borders of a block differ, the periodic extension has a jump, and a jump needs many high-frequency coefficients.

The discrete cosine transform

The (type-II) **discrete cosine transform** [3] uses real cosines:

$$t(u) = \alpha(u) \sum_{x=0}^{N-1} f(x) \cos\left[\frac{(2x+1)\, u\, \pi}{2N}\right], \qquad \alpha(0) = \sqrt{\tfrac{1}{N}},\ \ \alpha(u) = \sqrt{\tfrac{2}{N}} \text{ for } u \geq 1.$$

Here u is the frequency index and $\alpha(u)$ makes the basis orthonormal. The DCT equals (up to scaling) the DFT of a mirrored, even extension of the signal of length $2N$. Mirroring removes the border jump, so energy concentrates in fewer low-frequency coefficients. This **energy compaction** is the reason the JPEG standard applies an 8×8 DCT to image blocks [4]. Ahmed, Natarajan and Rao introduced the DCT in 1974 and showed that its performance compares closely with that of the Karhunen–Loève transform, the statistically optimal transform, in Wiener filtering and rate–distortion terms [3].

Online only: The discrete sine transform — https://blog.shyandram.dev/en/learn/image-processing/wavelets-transforms/notes/the-discrete-sine-transform/

Walsh–Hadamard transform

Plain version: build every basis vector from only $+1$ and -1. Then the transform needs no multiplications at all, just additions and subtractions.

The Hadamard matrix of order $N = 2^n$ is built recursively:

$$\mathbf{H}_1 = [\,1\,], \qquad \mathbf{H}_{2N} = \begin{bmatrix} \mathbf{H}_N & \mathbf{H}_N \\ \mathbf{H}_N & -\mathbf{H}_N \end{bmatrix}, \qquad \mathbf{A}_{\text{WHT}} = \frac{1}{\sqrt{N}} \mathbf{H}_N.$$

Every entry of $\mathbf{H}_N$ is ± 1, and its rows are mutually orthogonal, so $\mathbf{A}_{\text{WHT}}$ is orthonormal (and symmetric). The rows of $\mathbf{H}_N$ come out in **natural** (Hadamard) order. For analysis it is nicer to reorder them by **sequency**, the number of sign changes along a row, which plays the role of frequency. Sequency-ordered rows are called Walsh functions, and Figure 7.2 shows their 2-D basis images.

The Walsh–Hadamard transform (WHT) was used for early image coding because it is cheap [5]. Its energy compaction is weaker than the DCT's for natural images, because its square-wave basis approximates smooth shading poorly (see Figure 7.7 later). It remains useful where speed or hardware simplicity matters most, and as a building block for fast randomized algorithms.

```
import numpy as np
from scipy.linalg import hadamard

H = hadamard(8)                                  # natural (Hadamard) order
changes = (np.diff(np.sign(H), axis=1) != 0).sum(1)
W = H[np.argsort(changes)] / np.sqrt(8)          # sequency order
print((np.diff(np.sign(W), axis=1) != 0).sum(1))   # [0 1 2 3 4 5 6 7]

x = np.array([10, 12, 11, 13, 40, 42, 41, 43], float)
print(np.round(W @ x, 2))
```

The test signal is a step with small ripples. Its WHT is almost entirely in coefficient 0 (the mean) and coefficient 1 (a single sign change, matching the step); only a few small values remain.

Slant transform

Plain version: many image regions are not flat but get gradually brighter or darker. The slant transform includes a "ramp" building block so that such regions need only one or two coefficients.

Pratt, Chen and Welch designed the **slant transform** for image coding [6]. Like the WHT, it is orthonormal and built recursively: each size is made from two copies of the half-size matrix, mixed by a sparse matrix whose two constants are chosen so that the second basis vector is a uniformly decreasing staircase (the "slant" vector) and the matrix stays orthonormal. In Figure 7.2 the slant basis images in the first row and first column are visible ramps. The transform has a fast algorithm of $O(N \log_2 N)$ operations, but in practice the DCT, which also handles ramps well, displaced it.

Online only: Building the slant matrix — https://blog.shyandram.dev/en/learn/image-processing/wavelets-transforms/notes/building-the-slant-matrix/

Haar transform

Plain version: take two neighbors, write down their average and their difference. Repeat on the averages. That is the Haar transform, and it is the simplest wavelet.

The Haar transform [1] uses basis functions that are either constant or a single up-down step, at different widths and positions. For $N = 2^n$ samples, index the functions by a scale p and a position q, with $k = 2^p + q$ for $k \geq 1$. Up to normalization, the k-th basis vector equals $+1$ on the first half of the interval $\left[\frac{q}{2^p}, \frac{q+1}{2^p}\right)$, -1 on its second half, and 0 elsewhere (with positions measured as a fraction of the signal length). The $k = 0$ vector is constant. For $N = 4$,

$$\mathbf{A}_{\text{Haar}} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}.$$

Rows 0 and 1 look at the whole signal; rows 2 and 3 each look at only one half. That is the key difference from every transform above: **Haar basis functions are localized**. A local event, such as an edge, changes only the few Haar coefficients whose support covers it. In the time–frequency picture, Haar has exactly the dyadic tiling of Figure 7.1. It is the bridge to the wavelet transforms that follow: the Haar transform *is* the discrete wavelet transform with the Haar wavelet.

Wavelet transforms

Plain version: look at the image at many zoom levels. At each level, keep a blurrier copy and write down only the details that the blurring removed. The blurry copies form a pyramid; the details are the wavelet coefficients.

Multiresolution analysis

Multiresolution analysis (MRA), formalized by Mallat [7], is the framework that turns this idea into a basis. Start with a **scaling function** $\varphi(x)$ and form shifted and scaled copies,

$$\varphi_{j,k}(x) = 2^{j/2}\,\varphi(2^j x - k),$$

where j is the **scale** (larger j means narrower functions and finer detail), k is the integer shift, and $2^{j/2}$ keeps the energy at 1. Let V_j be the space spanned by $\{\varphi_{j,k}\}_k$: all signals representable at resolution j. MRA requires four things:

1. The $\varphi_{0,k}$ are orthonormal (or at least a stable basis of V_0).
2. The spaces are nested, $\cdots \subset V_{-1} \subset V_0 \subset V_1 \subset \cdots$: anything visible at a coarse scale is also visible at a finer one.
3. The only function common to all V_j is $f(x) = 0$.
4. Every square-integrable function can be approximated arbitrarily well as $j \to \infty$.

Because $V_0 \subset V_1$, the scaling function itself must be a combination of the finer ones. This gives the **refinement equation** (or dilation equation)

$$\varphi(x) = \sum_n h_\varphi(n)\,\sqrt{2}\,\varphi(2x - n),$$

where the coefficients $h_\varphi(n)$ are the **scaling function coefficients**. They will become a low-pass filter.

Wavelet functions

The details lost when going from V_{j+1} down to V_j live in a complementary space W_j, so that $V_{j+1} = V_j \oplus W_j$ (the symbol $\oplus$ means every element of V_{j+1} splits uniquely into a part in V_j and a part in W_j; for orthonormal wavelets the two parts are orthogonal). W_j is spanned by the **wavelet functions**

$$\psi_{j,k}(x) = 2^{j/2}\,\psi(2^j x - k), \qquad \psi(x) = \sum_n h_\psi(n)\,\sqrt{2}\,\varphi(2x - n).$$

For orthonormal wavelets, the **wavelet function coefficients** follow from the scaling coefficients by a modulation and time reversal, $h_\psi(n) = (-1)^n\,h_\varphi(1 - n)$. They form a high-pass filter. Repeating the split gives

$$L^2(\mathbb{R}) = V_{j_0} \oplus W_{j_0} \oplus W_{j_0+1} \oplus \cdots,$$

where $L^2(\mathbb{R})$ is the space of finite-energy functions: one coarse approximation plus details at every finer scale.

For Haar, $h_\varphi = [\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}]$ and $h_\psi = [\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}]$: average and difference. Daubechies showed how to construct orthonormal wavelets with **compact support** (finite-length filters) and increasing smoothness; the regularity grows linearly with the filter length [8]. These are the `dbN` families in PyWavelets [10], where `N` is the number of **vanishing moments**: $\int x^m \psi(x)\,dx = 0$ for $m = 0, \ldots, N - 1$. A wavelet with N vanishing moments ignores polynomials of degree below N, so smooth regions produce near-zero detail coefficients.

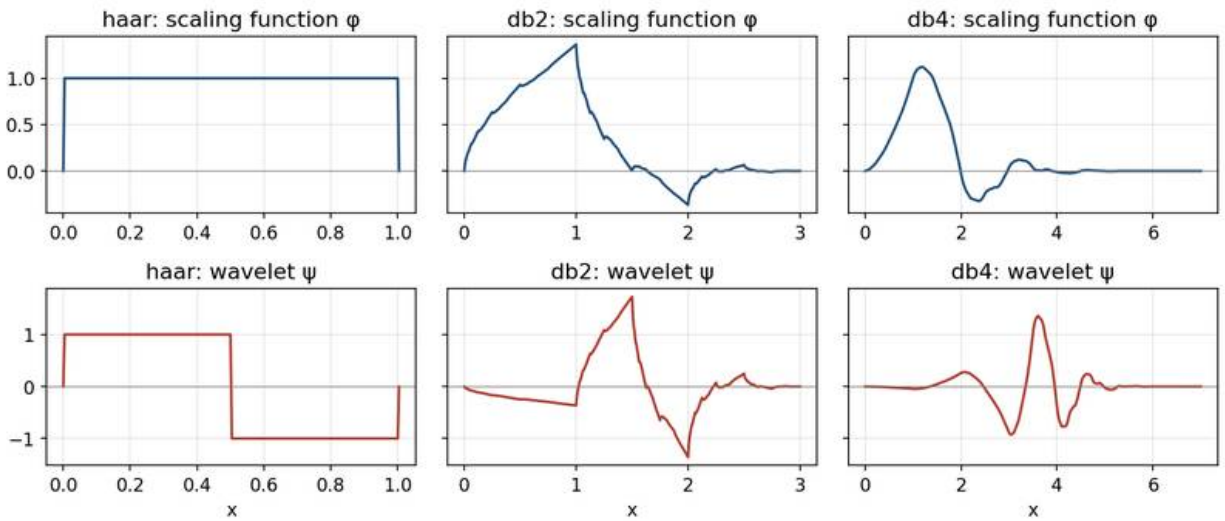


Figure 7.3 — Scaling functions (top) and wavelets (bottom) for Haar, db2 and db4, computed with PyWavelets' cascade algorithm. Longer filters give smoother, longer functions.

The discrete wavelet transform in one dimension

With these two families, any finite-energy signal splits into one coarse approximation plus details at every finer scale. The **approximation** (scaling) coefficients are inner products with the $\varphi_{j_0,k}$ at a chosen coarsest scale j_0, and the **detail** (wavelet) coefficients are inner products with the $\psi_{j,k}$ at every finer scale j, exactly as in the Preliminaries. For a sampled signal of length $M = 2^J$ the sums become finite, and the result is the **discrete wavelet transform** (DWT): approximation coefficients $W_\varphi(j_0, k)$ and detail coefficients $W_\psi(j, k)$, indexed by scale j and position k. For an orthonormal wavelet with a critically sampled finite-length boundary construction, the total number of coefficients equals M: this DWT is a change of basis, not an expansion. A library may retain additional boundary coefficients. In PyWavelets, `periodization` gives the smallest coefficient count, whereas its default symmetric extension can be redundant; use the same extension mode for analysis and synthesis [21].

Online only: The wavelet series and DWT formulas — https://blog.shyandram.dev/en/learn/image-processing/wavelets-transforms/notes/the-wavelet-series-and-dwt-formulas/

The fast wavelet transform

Plain version: you never need to evaluate φ or ψ. Two short filters and "keep every second sample" do all the work, level after level.

Substituting the refinement equations into the DWT gives Mallat's pyramid algorithm, now called the **fast wavelet transform** (FWT) [7]:

$$W_\psi(j,k) = \sum_n h_\psi(n-2k)\, W_\varphi(j+1,n), \qquad W_\varphi(j,k) = \sum_n h_\varphi(n-2k)\, W_\varphi(j+1,n).$$

Read it as: correlate the finer approximation $W_\varphi(j+1,\cdot)$ with h_ψ (or h_φ), then keep every second output (**downsampling by 2**). In signal-processing terms this is a two-channel **analysis filter bank**: a low-pass branch produces the next approximation and a high-pass branch produces the details. Feed the low-pass output into the same bank again, and so on. Each level halves the data, so the total cost is $O(M)$, faster than the FFT's $O(M \log M)$.

The inverse FWT is a **synthesis filter bank**: upsample each branch by 2 (insert zeros), filter with the synthesis filters, and add. For orthonormal wavelets the synthesis filters are the time-reversed analysis filters, and the bank achieves **perfect reconstruction**: the output equals the input exactly, despite the aliasing introduced by downsampling in each branch, because the aliasing terms of the two branches cancel. Vetterli and Kovačević's textbook develops this subband-coding view in depth [2].

```
import numpy as np
import pywt
```

```
x = np.array([4.0, 6.0, 10.0, 12.0, 8.0, 6.0, 5.0, 5.0])
w = pywt.Wavelet("haar")
h0, h1 = np.array(w.dec_lo), np.array(w.dec_hi)   # [.707 .707], [-.707 .707]

def analysis(x):
    lo = np.convolve(x, h0)[1::2]   # filter, then keep every 2nd sample
    hi = np.convolve(x, h1)[1::2]
    return lo, hi

a1, d1 = analysis(x)
a2, d2 = analysis(a1)
print("a2", a2, "d2", d2, "d1", d1.round(3))

ref = pywt.wavedec(x, "haar", level=2)            # [a2, d2, d1]
print(all(np.allclose(p, q) for p, q in zip(ref, [a2, d2, d1])))
print(np.allclose(pywt.waverec(ref, "haar"), x))
```

The two-line filter bank reproduces `pywt.wavedec` exactly. Note how the detail coefficient is 0 for the pair $(5, 5)$: no change, no detail.

Wavelet transforms in two dimensions

Plain version: run the 1-D filter bank along the rows, then along the columns. You get one small blurry image and three "edge maps", one per direction.

The 2-D DWT is separable. One level uses a 2-D scaling function and three 2-D wavelets, each a product of 1-D functions:

$$\varphi(x,y) = \varphi(x)\varphi(y), \quad \psi^H(x,y) = \psi(x)\varphi(y), \quad \psi^V(x,y) = \varphi(x)\psi(y), \quad \psi^D(x,y) = \psi(x)\psi(y).$$

Here x is the row index (vertical position) and y the column index, following the book's convention. ψ^H changes along the vertical direction and is smooth horizontally, so it responds to **horizontal** edges. ψ^V responds to **vertical** edges, and ψ^D to diagonal detail and corners. One level turns an $M \times N$ image into four $\frac{M}{2} \times \frac{N}{2}$ **subbands**:

Subband	Rows filter	Columns filter	Often called	PyWavelets name
Approximation	low	low	LL	`cA`
Horizontal detail	high	low	LH or HL (conventions differ)	`cH`
Vertical detail	low	high	HL or LH	`cV`
Diagonal detail	high	high	HH	`cD`

"Rows filter" here means the filter applied across rows, that is, along the vertical direction. Because the LH/HL labels are swapped between textbooks and libraries, this chapter uses the unambiguous names H, V and D. Repeating the decomposition on the approximation band gives the familiar nested layout in Figure 7.4.

```
import numpy as np
import pywt
from skimage import data, img_as_float

img = img_as_float(data.camera())
LL1, (H1, V1, D1) = pywt.dwt2(img, "haar")   # one level
LL2, (H2, V2, D2) = pywt.dwt2(LL1, "haar")   # second level on LL1
print(img.shape, LL1.shape, LL2.shape)       # (512, 512) (256, 256) (128, 128)

e = lambda a: float((a ** 2).sum())
total = e(img)
print(f"energy in LL2: {e(LL2) / total:.4f}")

coeffs = pywt.wavedec2(img, "haar", level=2)  # same thing in one call
print(np.allclose(pywt.waverec2(coeffs, "haar"), img))
```

After two levels, the 128 × 128 approximation, one sixteenth of the coefficients, holds about 99% of the energy of the camera image. The other 15/16 of the coefficients are mostly near zero, which is exactly the sparsity that compression and denoising exploit.

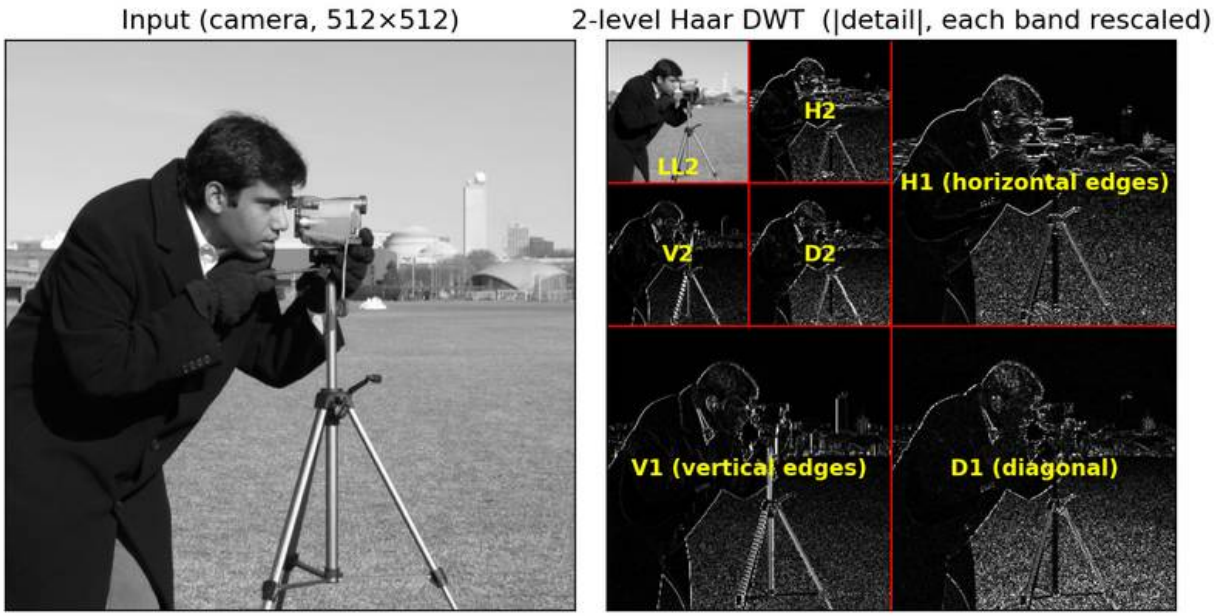


Figure 7.4 — Two-level Haar DWT of the camera image. The detail bands show absolute values, each rescaled to its own 99.5th percentile so that the small coefficients are visible; on a common scale they would look almost black. The tripod's vertical legs light up in V1; horizontal outlines of the buildings show in H1.

Online only: Wavelet packets — https://blog.shyandram.dev/en/learn/image-processing/wavelets-transforms/notes/wavelet-packets/

Applications

Denoising by thresholding

Plain version: in the wavelet domain, a clean image is a few large numbers, while white noise is many small numbers spread everywhere. Set the small ones to zero and transform back.

An orthonormal transform maps white Gaussian noise of standard deviation σ to white Gaussian noise of the same σ in every subband, while the image's energy concentrates in few coefficients. Thresholding rules for a detail coefficient w and threshold T:

$$\eta_{\text{hard}}(w) = \begin{cases} w, & |w| > T \\ 0, & |w| \le T \end{cases} \qquad \eta_{\text{soft}}(w) = \operatorname{sign}(w)\ \max(|w| - T,\, 0).$$

Hard thresholding keeps or kills. Soft thresholding also shrinks the survivors toward zero by T, which gives a continuous rule with fewer artifacts but some loss of contrast. Two classic choices of T:

- **Universal threshold** (VisuShrink), $T = \sigma\sqrt{2\ln n}$ for n samples, from Donoho and Johnstone [12]. Donoho showed that soft thresholding with this T gives, with high probability, an estimate that is at least as smooth as the true signal [13]. The noise level is usually estimated robustly from the finest diagonal band as $\hat{\sigma} = \text{median}(|w|)/0.6745$ [12].
- **BayesShrink**, from Chang, Yu and Vetterli, sets a separate threshold for each subband, $T = \hat{\sigma}^2/\hat{\sigma}_X$, where $\hat{\sigma}_X$ is the estimated standard deviation of the clean coefficients in that band [14]. It adapts to how much signal each band contains and usually beats the universal threshold.

```
import numpy as np
import pywt
from skimage import data, img_as_float
from skimage.metrics import peak_signal_noise_ratio as psnr

rng = np.random.default_rng(7)
clean = img_as_float(data.camera())
noisy = clean + rng.normal(0, 0.1, clean.shape)

def wavelet_denoise(y, wavelet="db4", level=3, mode="soft"):
    c = pywt.wavedec2(y, wavelet, mode="periodization", level=level)
    sigma = np.median(np.abs(c[-1][2])) / 0.6745       # noise level from finest HH
    T = sigma * np.sqrt(2 * np.log(y.size))            # universal threshold
    c = [c[0]] + [tuple(pywt.threshold(d, T, mode) for d in lvl) for lvl in c[1:]]
    return pywt.waverec2(c, wavelet, mode="periodization")

for mode in ["hard", "soft"]:
    out = np.clip(wavelet_denoise(noisy, mode=mode), 0, 1)
    print(mode, round(psnr(clean, out, data_range=1), 1), "dB")
print("noisy", round(psnr(clean, np.clip(noisy, 0, 1), data_range=1), 1), "dB")
```

On this example the noisy image scores 20.4 dB PSNR, hard thresholding 25.7 dB and soft thresholding 24.7 dB. The universal threshold is conservative (it is designed to remove essentially all

the noise), so with soft shrinkage on top it over-smooths; the cartoon-like look in Figure 7.5 is the price. Subband-adaptive thresholds such as BayesShrink, or an undecimated (shift-invariant) transform, reduce both the blur and the blocky artifacts. scikit-image wraps these variants in `skimage.restoration.denoise_wavelet`.

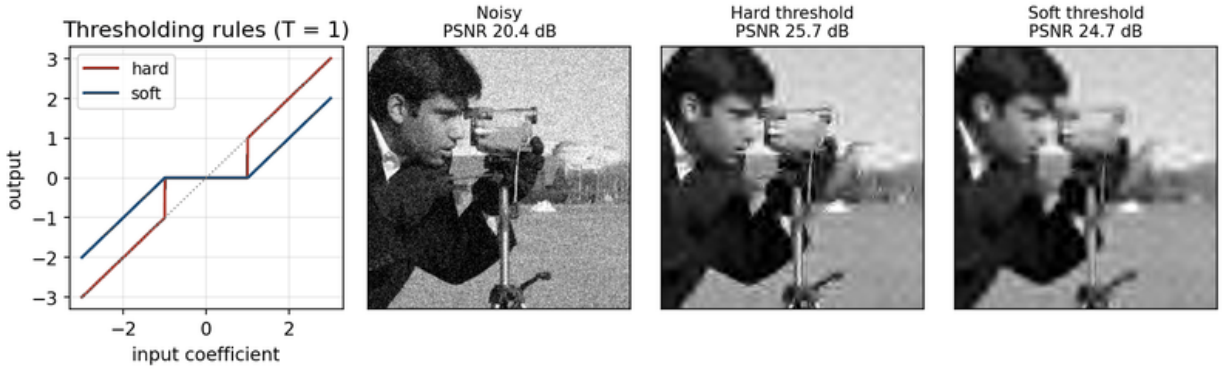
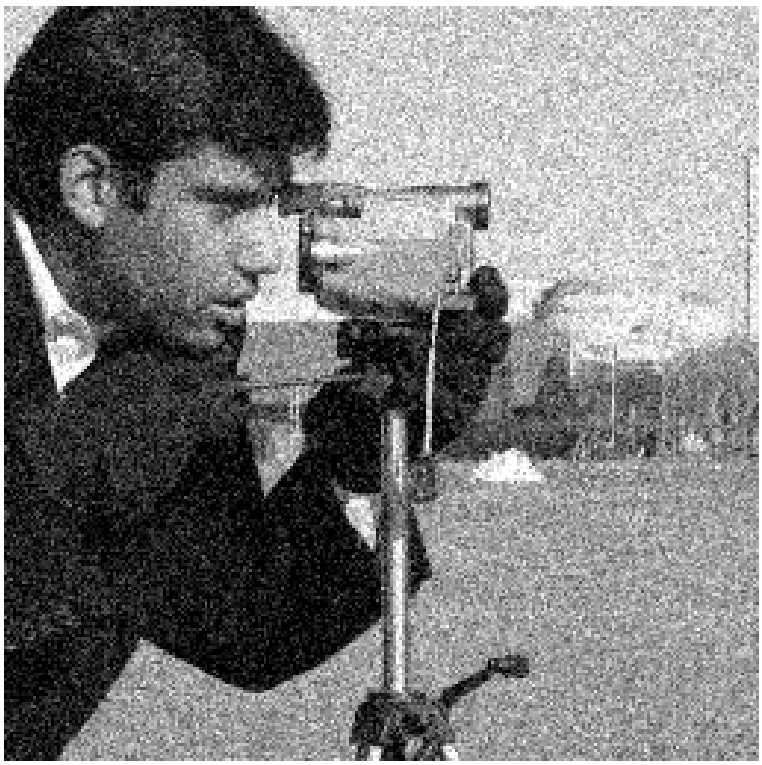


Figure 7.5 — Wavelet denoising of the camera image (Gaussian noise, $\sigma = 0.1$ on a $[0, 1]$ scale) with db4, three levels and the universal threshold. Hard thresholding keeps sharper edges but leaves some isolated blips; soft thresholding is smoother but blurrier.

Drag the slider to compare the noisy input with the soft-thresholded result:



Noisy



Soft threshold

Noisy and wavelet-denoised crops of the camera image

Edge detection

Because the detail bands are high-pass responses in three directions, wavelets give a quick edge detector: set the approximation band to zero and invert the transform. What remains is the image minus its coarse version, which is concentrated at edges. Keeping only some detail bands selects edge orientation, as in Figure 7.6. Coarser levels give thicker but more noise-robust edges, a multiscale behavior similar to choosing the σ of a Gaussian-derivative edge detector.

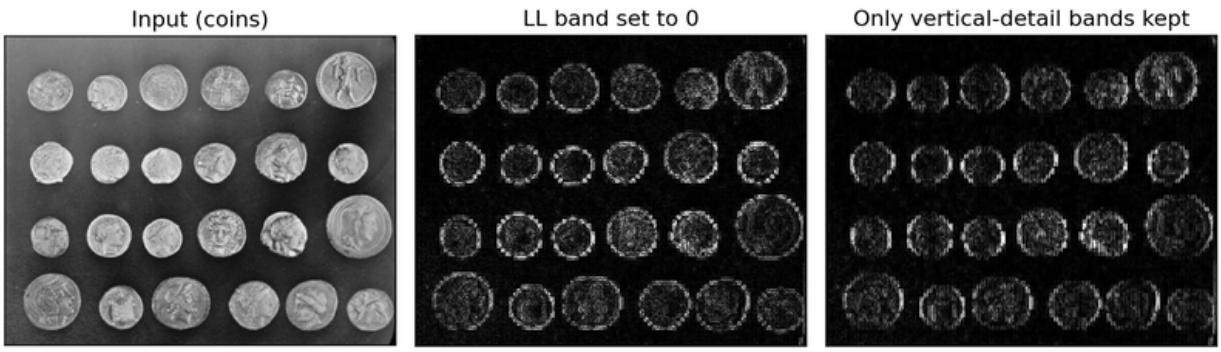


Figure 7.6 — Edges from a two-level Haar DWT of the coins image. Middle: approximation band set to zero. Right: only the vertical-detail bands kept, which emphasizes the left and right rims of each coin.

Compression preview

Compression rests on energy compaction: transform, keep the few big coefficients, and code them efficiently. Figure 7.7 keeps only the largest 5% of coefficients of the whole 512 × 512 image for four transforms. The two wavelets win by more than 2 dB over the DCT and more than 4 dB over the WHT, and their errors look different: the Fourier-type bases ring around edges and lose texture everywhere, while wavelets keep edges crisp and lose detail mainly in smooth or textured regions.

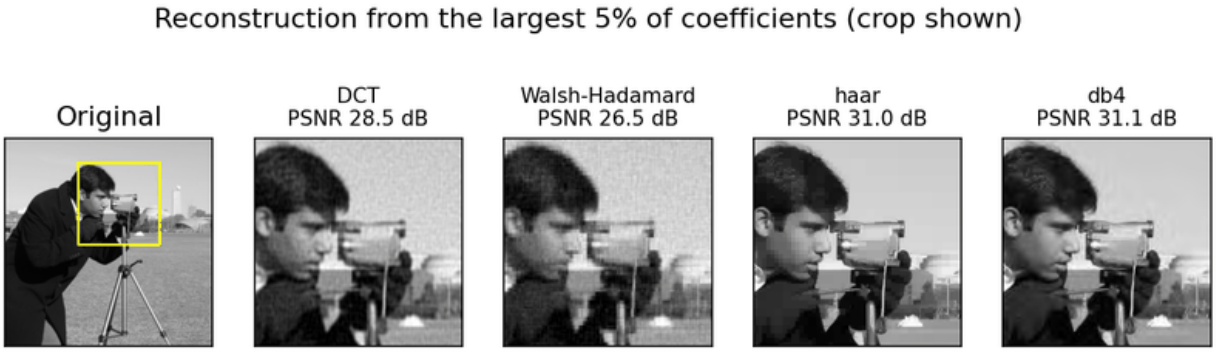


Figure 7.7 — Reconstructions from the largest 5% of coefficients (whole-image transforms, crop shown). PSNR: DCT 28.5 dB, Walsh–Hadamard 26.5 dB, Haar 31.0 dB, db4 31.1 dB.

Real codecs are more careful than "keep the top 5%". JPEG uses a block DCT with quantization and entropy coding [4], and JPEG 2000 uses a multilevel 2-D DWT with biorthogonal filters, followed by bit-plane coding of each subband [15]. IP 08 covers these pipelines in detail, and its transform-coding lab lets you quantize the DCT and Haar coefficients of a photo and see where the error goes.

Modern view

Wavelets arrived in image processing in the late 1980s as a unifying theory, peaked in the 1990s and 2000s as the backbone of denoising and JPEG 2000, and are now mostly an *ingredient*: of sparse models, of signal-processing-inspired network layers, and of efficient generative models. Here are the surveys and landmark papers worth reading, and what each contributes.

Online only: The Modern view in more detail — https://blog.shyandram.dev/en/learn/image-processing/wavelets-transforms/papers/the-modern-view-in-more-detail/

- **Foundations.** Mallat [7] introduced multiresolution analysis for images, the link to two-channel filter banks and the $O(N)$ pyramid algorithm; Daubechies [8] supplied orthonormal wavelets with compact support, which made the filters short and practical.

- **Surveys.** Vetterli and Kovačević's freely available textbook [2] develops the filter-bank view of this chapter; Unser and Blu [9] explain where vanishing moments, approximation order and smoothness come from; Rubinstein, Bruckstein and Elad [16] trace the move from analytic transforms to dictionaries learned from data.
- **Practice.** Skodras, Christopoulos and Ebrahimi [15] show how the DWT, biorthogonal filters, subband quantization and embedded coding fit together in JPEG 2000.
- **Denoising theory.** The shrinkage papers [12], [13], [14] showed that a simple nonlinearity in the wavelet domain is close to optimal over broad classes of signals, and that adapting the threshold per subband helps in practice.

What deep learning changed, and what it did not

- **Scattering networks.** Bruna and Mallat's **wavelet scattering transform** [17] is a convolutional network whose filters are fixed wavelets, followed by a modulus nonlinearity and averaging. It is provably stable to small deformations and became a reference point for explaining *why* CNNs work.
- **Wavelets as down-samplers.** Strided convolution and max-pooling downsample without proper low-pass filtering, so they alias. WaveCNet [19] replaces them with DWT layers in classifiers, which improved robustness to noise, and **MWCNN** [18] replaces the pooling of a U-Net with the invertible DWT, so no information is lost when the feature maps shrink.
- **Large receptive fields.** The **WTConv** layer [20] (ECCV 2024) applies small convolutions to the subbands of a multilevel DWT, so the receptive field grows quickly with the number of levels at little parameter cost.

What did not change. The DCT still sits inside JPEG, and the DCT and its integer approximations remain core tools in conventional image and video codecs. Hand-tuned wavelet thresholding has been overtaken in denoising quality by trained networks, but it remains a useful baseline that needs no training data. Its risk guarantees depend on the signal and noise assumptions of the particular shrinkage result; speed depends on the implementation and image size. Most importantly, the vocabulary of this chapter (bases, frames, filter banks, multiresolution, aliasing, sparsity) is the vocabulary used to analyze what convolutional networks do.

Key takeaways

- A linear transform is a change of basis. Each coefficient is an inner product, that is, a correlation of the image with one basis function.
- Orthonormal transforms are matrix products, $\mathbf{T} = \mathbf{AFA}^T$ for separable 2-D cases, with the transpose as the inverse and energy preserved.
- The DCT compacts the energy of smooth images better than the DFT or WHT because its implied even extension has no border jump; this is why JPEG uses it.
- WHT and slant transforms trade some compaction for simplicity (only ± 1) or for an explicit ramp vector; Haar adds localization.

- The time–frequency plane shows the trade-off: Fourier bases are sharp in frequency, samples sharp in time, wavelets adapt the tile shape to the frequency.
- Multiresolution analysis turns two short filters into a fast $O(N)$ wavelet transform. In 2-D it yields an approximation band and horizontal, vertical and diagonal detail bands at each level.
- Sparsity in the wavelet domain powers denoising (hard/soft thresholding), quick edge maps and compression (JPEG 2000).
- Modern networks reuse wavelets as fixed feature extractors (scattering), as invertible downsamplers (MWCNN, WaveCNet) and as cheap large-receptive-field layers (WTConv).

Exercises

1. **A frame by hand.** Take the three unit vectors in the plane at angles 90°, 210° and 330°. Compute $\sum_k \langle \mathbf{f}, \mathbf{s}_k \rangle^2$ for $\mathbf{f} = (1, 0)$ and for $\mathbf{f} = (0, 1)$. What are the frame bounds, and how do you reconstruct $\mathbf{f}$ from its three coefficients?

   Hint

   Both sums equal 3/2; in fact, for every unit $\mathbf{f}$ the sum is 3/2, so the frame is tight with $A = B = 3/2$. Reconstruct with $\mathbf{f} = \frac{2}{3} \sum_k \langle \mathbf{f}, \mathbf{s}_k \rangle \mathbf{s}_k$.

2. **Separable cost.** For a 256×256 image, count the multiplications for (a) a general 2-D linear transform written as one 65536×65536 matrix, (b) the separable form $\mathbf{A}\mathbf{F}\mathbf{A}^T$ with dense $\mathbf{A}$, and (c) a three-level Haar FWT.

   Hint

   (a) $N^4 \approx 4.3 \times 10^9$. (b) two products of N^3 each, $2N^3 \approx 3.4 \times 10^7$. (c) Each level filters every row and column of the current approximation with 2-tap filters at half rate; the total is a small constant times N^2, a few hundred thousand.

3. **Why mirror?** Take $f = [1, 2, 3, 4, 5, 6, 7, 8]$ (a ramp). Compute its DFT and its DCT with SciPy and count how many coefficients you need to capture 99% of the energy in each. Explain the difference using the implied periodic and even extensions.

   Hint

   The periodic extension jumps from 8 back to 1, which needs many DFT harmonics. The even extension $\ldots 2, 1, 1, 2, \ldots, 8, 8, 7, \ldots$ is continuous, so the DCT needs only two or three coefficients.

4. **Haar by hand.** Compute a two-level Haar DWT of $[2, 2, 6, 6, 3, 5, 9, 9]$ using averages and differences scaled by $1/\sqrt{2}$. Which detail coefficients are zero, and why? Check your answer with `pywt.wavedec`.

Hint

Level 1 pairs: (2,2), (6,6), (3,5), (9,9). Only the pair (3,5) has a non-zero difference. At level 2, compare the scaled sums of neighboring pairs. Watch PyWavelets' sign convention: with its Haar filters, each detail coefficient comes out as (first − second)/$\sqrt{2}$, so the pair (3, 5) gives $-\sqrt{2}$.

5. **Thresholds in practice.** Modify the denoising code to use (a) a threshold equal to $3\hat{\sigma}$, and (b) a per-subband BayesShrink threshold $\hat{\sigma}^2/\hat{\sigma}_X$ with $\hat{\sigma}_X = \sqrt{\max(\overline{w^2} - \hat{\sigma}^2, 0)}$ computed in each band. Compare PSNR with the universal threshold for $\sigma = 0.05$ and $\sigma = 0.2$.

Hint

The universal threshold for a 512×512 image is about $5\hat{\sigma}$, so (a) keeps more coefficients. Expect (b) to be the best of the three for soft thresholding. If $\hat{\sigma}_X = 0$, set the band to zero.

6. **Shift sensitivity.** Shift the camera image by one pixel horizontally, take a one-level Haar DWT of both versions, and compare the energy in the V band. Why does a one-pixel shift change the coefficients so much, and how would an undecimated transform (`pywt.swt2`) behave?

Hint

Downsampling by 2 makes the DWT shift-variant: an edge that falls inside a Haar pair produces a detail coefficient, while the same edge falling between pairs produces none. The stationary (undecimated) transform skips downsampling, so it is a redundant frame and shifts its coefficients along with the image.

References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018, Ch. 7. publisher page
2. M. Vetterli and J. Kovačević, *Wavelets and Subband Coding*, Prentice Hall, 1995 (free edition from the authors). book site
3. N. Ahmed, T. Natarajan and K. R. Rao, "Discrete Cosine Transform," *IEEE Transactions on Computers*, 1974. doi
4. G. K. Wallace, "The JPEG still picture compression standard," *IEEE Transactions on Consumer Electronics*, 1992. doi
5. W. K. Pratt, J. Kane and H. C. Andrews, "Hadamard transform image coding," *Proceedings of the IEEE*, 1969. doi
6. W. K. Pratt, W.-H. Chen and L. R. Welch, "Slant transform image coding," *IEEE Transactions on Communications*, 1974. doi
7. S. G. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1989. doi
8. I. Daubechies, "Orthonormal bases of compactly supported wavelets," *Communications on Pure and Applied Mathematics*, 1988. doi

9. M. Unser and T. Blu, “Wavelet theory demystified,” *IEEE Transactions on Signal Processing*, 2003. doi
10. G. R. Lee, R. Gommers, F. Waselewski, K. Wohlfahrt and A. O’Leary, “PyWavelets: A Python package for wavelet analysis,” *Journal of Open Source Software*, 2019. doi
11. R. R. Coifman and M. V. Wickerhauser, “Entropy-based algorithms for best basis selection,” *IEEE Transactions on Information Theory*, 1992. doi
12. D. L. Donoho and I. M. Johnstone, “Ideal spatial adaptation by wavelet shrinkage,” *Biometrika*, 1994. doi
13. D. L. Donoho, “De-noising by soft-thresholding,” *IEEE Transactions on Information Theory*, 1995. doi
14. S. G. Chang, B. Yu and M. Vetterli, “Adaptive wavelet thresholding for image denoising and compression,” *IEEE Transactions on Image Processing*, 2000. doi
15. A. Skodras, C. Christopoulos and T. Ebrahimi, “The JPEG 2000 still image compression standard,” *IEEE Signal Processing Magazine*, 2001. doi
16. R. Rubinstein, A. M. Bruckstein and M. Elad, “Dictionaries for sparse representation modeling,” *Proceedings of the IEEE*, 2010. doi
17. J. Bruna and S. Mallat, “Invariant scattering convolution networks,” arXiv:1203.1513, 2012. arXiv
18. P. Liu, H. Zhang, K. Zhang, L. Lin and W. Zuo, “Multi-level wavelet-CNN for image restoration,” CVPR Workshops (NTIRE), 2018. arXiv
19. Q. Li, L. Shen, S. Guo and Z. Lai, “Wavelet integrated CNNs for noise-robust image classification,” CVPR, 2020. arXiv
20. S. E. Finder, R. Amoyal, E. Treister and O. Freifeld, “Wavelet convolutions for large receptive fields,” ECCV, 2024. arXiv
21. PyWavelets developers, “Signal extension modes,” PyWavelets documentation, accessed 5 November 2026. Documentation.

IP 08 • Image Compression and Watermarking

How images are squeezed into fewer bits, from entropy and Huffman codes to JPEG, JPEG 2000 and learned codecs, and how hidden marks are written into them.

Online edition: https://blog.shyandram.dev/en/learn/image-processing/compression-watermarking/

What you'll learn

- The three kinds of redundancy that compression removes, and how entropy puts a floor under lossless coding
- How the classic lossless coders work: Huffman, Golomb, arithmetic, LZW, run-length, symbol-based and bit-plane coding
- How JPEG turns an image into 8×8 DCT blocks, quantizes them, and why it shows blocks at low quality
- How predictive coding (DPCM, motion compensation) and wavelet coding (JPEG 2000) exploit neighboring pixels
- How to hide a watermark in the least significant bits or in DCT coefficients, and why one survives JPEG and the other does not
- Where learned (neural) codecs and deep-learning watermarks fit into this picture today

The big picture

A raw 512×512 color photo is 786,432 bytes. The same photo as a JPEG can be a few tens of kilobytes and still look fine. Compression is the art of finding out which bits are predictable or invisible, and then not storing them.

> **Explain like I'm five**
>
> Imagine telling a friend over the phone what a picture of the sky looks like. You would not say "blue, blue, blue, blue..." a million times. You would say "the top is all blue, and there is one white cloud on the left." That is compression: say the common things briefly, and only spend words on the surprises. Watermarking is like writing your name on the drawing in invisible ink, so that later you can prove it is yours.

Nearly every image you see has been compressed, and many vision datasets are stored as JPEG, so models learn on images that already carry compression artifacts. Watermarking matters for ownership and, increasingly, for marking images made by generative models. Following the textbook [1], we cover fundamentals, then a toolbox of coding methods, then watermarking.

Fundamentals

In plain words: an image file usually contains more bits than the information it carries. Compression removes the extra bits; a good measure of "information" tells us how far we can go.

Compression ratio and redundancy

If an image takes b bits uncompressed and b' bits compressed, the **compression ratio** and **relative data redundancy** are

$$C = \frac{b}{b'}, \qquad R = 1 - \frac{1}{C}.$$

Here C is a pure number (10 means "ten to one") and R is the fraction of the original data that was redundant. With $C = 10$, $R = 0.9$: 90% of the bits carried no new information.

Images contain three kinds of redundancy:

- **Coding redundancy.** A fixed 8-bit code spends 8 bits on every gray level, common or rare. Variable-length codes waste fewer bits.
- **Spatial and temporal redundancy.** Neighboring pixels, and consecutive video frames, are strongly correlated, so coding each pixel independently repeats information.
- **Irrelevant information.** Detail a viewer does not notice. Removing it is irreversible (**lossy** compression); removing only the first two kinds is **lossless**.

Measuring information: entropy

A symbol that occurs with probability $P(a)$ carries $I(a) = -\log_2 P(a)$ bits of information: rare events are surprising and carry more bits. Averaging over the source alphabet gives the **entropy**

$$H = -\sum_{j=1}^{J} P(a_j)\,\log_2 P(a_j) \quad \text{bits/symbol},$$

where $a_1, \ldots, a_J$ are the possible symbols (for an 8-bit image, the 256 gray levels) and $P(a_j)$ is their probability. The noiseless source coding theorem says that no lossless code can use fewer than H bits per symbol on average when symbols are coded one at a time and independently; good codes come close to H.

For a real image we only have estimates. The **first-order estimate** uses the normalized histogram $p_r(r_k) = n_k/MN$, where n_k counts pixels with gray level r_k in an $M \times N$ image. Figure 8.1 shows why the estimate depends on what you call a "symbol". The camera image has a first-order entropy of 7.23 bits/pixel, barely below 8. If we instead code the difference between each pixel and its left neighbor, the entropy drops to 4.70 bits. The differences are not more clever; they simply remove the spatial redundancy that the histogram cannot see. This is the idea behind predictive coding later in the chapter.

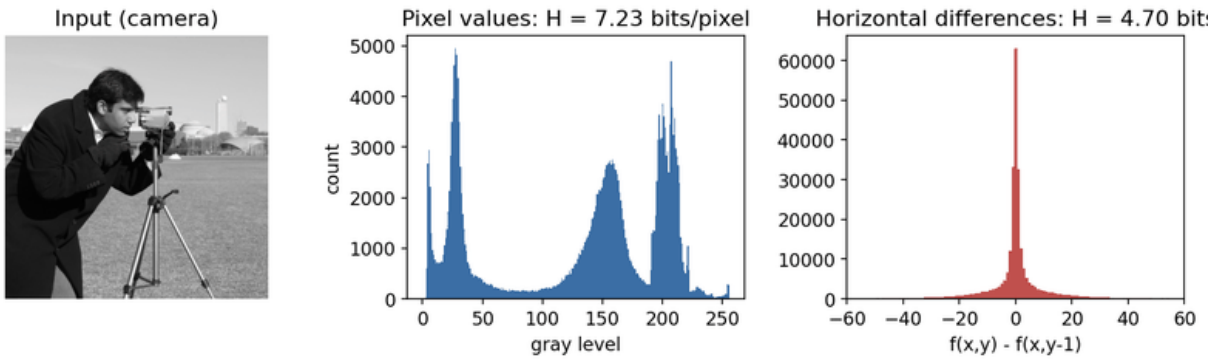


Figure 8.1 — The same image, two symbol choices. Pixel values are spread out (7.23 bits); differences between neighbors pile up near zero (4.70 bits).

```
import numpy as np
from skimage import data

def entropy(x):
    _, counts = np.unique(x, return_counts=True)
    p = counts / counts.sum()
    return -(p * np.log2(p)).sum()

img = data.camera()
print(f"H(pixels) = {entropy(img):.2f} bits")                          # 7.23
print(f"H(diffs)  = {entropy(np.diff(img.astype(int), axis=1)):.2f} bits")  # 4.70
```

Fidelity criteria

When compression is lossy we need to measure the damage. Let $f(x,y)$ be the original and $\hat{f}(x,y)$ the decompressed $M \times N$ image. The **root-mean-square error** and the **peak signal-to-noise ratio** are

$$e_{\text{rms}} = \sqrt{\frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}\left[\hat{f}(x,y)-f(x,y)\right]^2}, \qquad \text{PSNR} = 20\log_{10}\frac{L_{\max}}{e_{\text{rms}}}\ \text{dB},$$

where $L_{\max}$ is the largest possible pixel value (255 for 8-bit images). Higher PSNR means smaller error; as a rough guide, differences near 40 dB are hard to see and those below 25 dB are obvious, though this depends on content.

These **objective** criteria are easy to compute but do not always agree with people. **Subjective** criteria ask observers to rate images on a scale (from "excellent" to "unusable") or to compare pairs.

Image compression models

Almost every codec in this chapter fits one block diagram. The **encoder** has three stages:

1. a **mapper** that transforms the image into a form with less redundancy (differences, transform coefficients, run lengths);
2. a **quantizer** that discards precision according to a fidelity target (this is the only lossy stage, and lossless codecs omit it);
3. a **symbol coder** that assigns short codes to frequent symbols (Huffman, arithmetic, Golomb…).

The **decoder** runs a symbol decoder and an inverse mapper. There is no "inverse quantizer" that can bring back lost precision.

Online only: Formats, containers and standards — https://blog.shyandram.dev/en/learn/image-processing/compression-watermarking/notes/formats-containers-and-standards/

Huffman coding

In plain words: give the most common symbols the shortest codewords, and build the code so no codeword is the beginning of another, so the bit stream can be read without separators.

Huffman's algorithm [8] builds an optimal **prefix code** for a known set of symbol probabilities, when symbols are coded one at a time. It works bottom-up:

1. List the symbols with their probabilities.
2. Merge the two least probable entries into a new node whose probability is their sum.
3. Repeat until one node (probability 1) remains.
4. Walk from the root to each leaf, writing 0 for one branch and 1 for the other. The path is the codeword.

A worked example

Take a source with six symbols: $P(a_1) = 0.40$, $P(a_2) = 0.20$, $P(a_3) = 0.15$, $P(a_4) = 0.10$, $P(a_5) = 0.10$, $P(a_6) = 0.05$. The merges are:

Step	Merge	New node
1	a_6 (0.05) + a_4 (0.10)	0.15
2	a_5 (0.10) + a_3 (0.15)	0.25
3	a_2 (0.20) + node 0.15	0.35
4	node 0.25 + node 0.35	0.60
5	a_1 (0.40) + node 0.60	1.00

The tree in Figure 8.2 gives \$a_1 = \$ `1`, \$a_2 = \$ `000`, \$a_3 = \$ `010`, \$a_4 = \$ `0010`, \$a_5 = \$ `011`, \$a_6 = \$ `0011`. The average codeword length is

$$L_{\text{avg}} = \sum_{j} P(a_j)\, l(a_j) = 0.4(1) + 0.2(3) + 0.15(3) + 0.1(4) + 0.1(3) + 0.05(4) = 2.35 \text{ bits/symbol},$$

where $l(a_j)$ is the length of the codeword for a_j. A fixed-length code needs 3 bits and the entropy is $H = 2.284$ bits, so Huffman is within 3% of the bound. Ties can be broken either way; other choices change the codewords but not the average length.

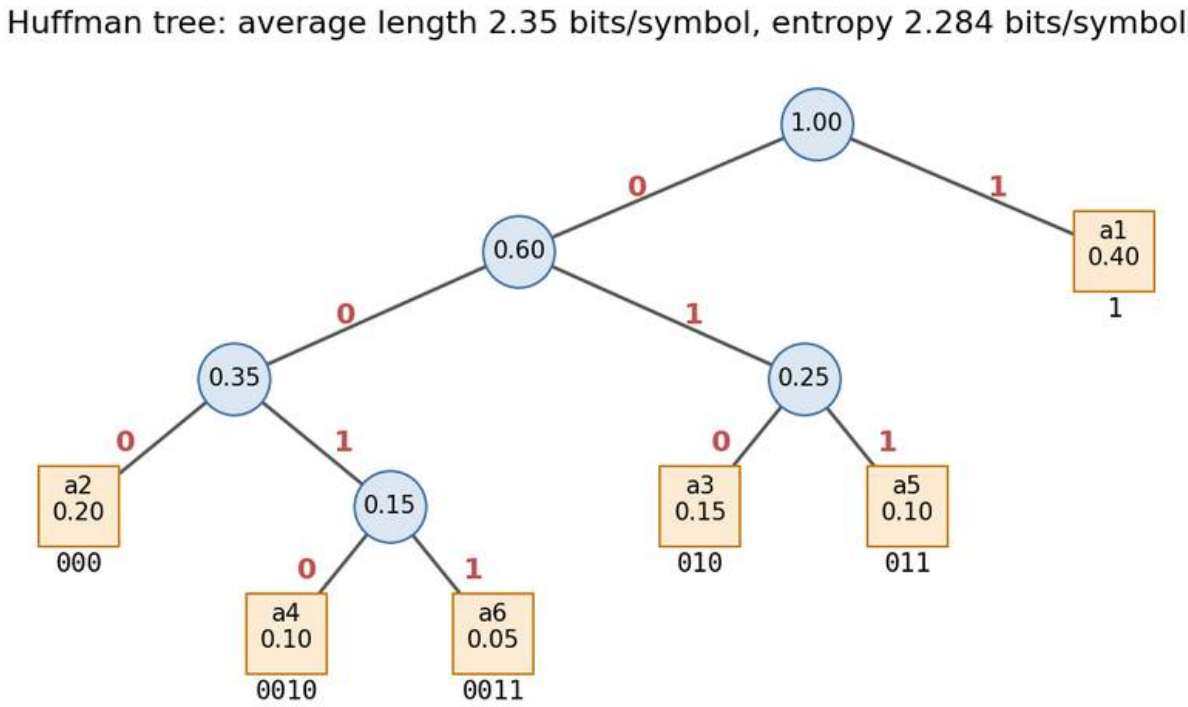


Figure 8.2 — Huffman tree for the six-symbol example. Edge labels give the bits; codewords are printed under each leaf.

Decoding is a walk down the tree: read bits until you hit a leaf, output its symbol, return to the root. Because no codeword is a prefix of another, the code is **instantaneous** and **uniquely decodable**. JPEG uses Huffman tables for its quantized coefficients [2].

```
import heapq, itertools

def huffman_codes(probs):
    tick = itertools.count()                       # tie-breaker for equal probabilities
    heap = [(p, next(tick), {s: ""}) for s, p in probs.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        p1, _, c1 = heapq.heappop(heap)            # least probable
        p2, _, c2 = heapq.heappop(heap)            # second least
        merged = {s: "1" + c for s, c in c1.items()}
        merged |= {s: "0" + c for s, c in c2.items()}
        heapq.heappush(heap, (p1 + p2, next(tick), merged))
    return heap[0][2]

P = {"a1": .40, "a2": .20, "a3": .15, "a4": .10, "a5": .10, "a6": .05}
print(huffman_codes(P))   # a1:'1', a2:'000', a3:'010', a4:'0010', a5:'011', a6:'0011'
```

The limitation: every symbol costs at least one whole bit. When one symbol has probability 0.95, its information is only 0.07 bits, but Huffman still spends 1 bit on it. Arithmetic coding, below, removes that limit.

Golomb coding

In plain words: when small numbers are common and big numbers are rare, write a number as "how many full groups of m" (in tally marks) plus "what is left over" (in ordinary binary).

Golomb codes [9] are designed for nonnegative integers n with a geometric-like distribution, which is exactly what prediction residuals look like after mapping. For a parameter $m > 0$, the code $G_m(n)$ is the concatenation of

1. the **unary** code of the quotient $q = \lfloor n/m \rfloor$: q ones followed by a zero;
2. the **truncated binary** code of the remainder $r = n \bmod m$: with $k = \lceil \log_2 m \rceil$ and $c = 2^k - m$, values $r < c$ use $k - 1$ bits, and the rest are coded as $r + c$ in k bits.

n	0	1	2	3	4	5	6	7	8
$G_4(n)$	000	001	010	011	1000	1001	1010	1011	11000
$G_3(n)$	00	010	011	100	1010	1011	1100	11010	11011

When m is a power of two, the remainder is just the low k bits of n, so encoding needs only shifts and masks. Prediction residuals e are signed, so a codec first maps them to nonnegative integers, for example $M(e) = 2e$ for $e \geq 0$ and $M(e) = 2|e| - 1$ for $e < 0$, which interleaves $0, -1, 1, -2, 2, \ldots$. JPEG-LS [10] combines this mapping with Golomb codes whose power-of-two parameter is adapted to local context.

```
import numpy as np

def golomb(n, m):
    q, r = divmod(n, m)
    unary = "1" * q + "0"
    k = int(np.ceil(np.log2(m)))
    c = 2**k - m                        # how many remainders get k-1 bits
    if r < c:
        tail = format(r, f"0{k-1}b") if k > 1 else ""
    else:
        tail = format(r + c, f"0{k}b")
    return unary + tail

print([golomb(n, 3) for n in range(9)])
```

Arithmetic coding

In plain words: instead of giving each symbol its own codeword, describe the whole message as one very precise number between 0 and 1.

Arithmetic coding [11] starts from the interval $[0, 1)$. For each symbol, it shrinks the current interval to the sub-interval that the symbol owns, in proportion to its probability. After the message, any number inside the final interval identifies the message. If the current interval is $[\ell, h)$ and the next symbol s owns the cumulative range $[F_{\text{lo}}(s), F_{\text{hi}}(s))$, then

$$\ell' = \ell + (h - \ell)\, F_{\text{lo}}(s), \qquad h' = \ell + (h - \ell)\, F_{\text{hi}}(s).$$

The final width equals the product of the symbol probabilities, so the number of bits needed is about $-\log_2 \prod_i P(s_i) = \sum_i -\log_2 P(s_i)$, which is exactly the information content of the message. There is no "one bit per symbol" floor.

Worked example

Let $P(a) = 0.6$, $P(b) = 0.3$, $P(c) = 0.1$, so a owns $[0, 0.6)$, b owns $[0.6, 0.9)$ and c owns $[0.9, 1)$. Encode the message `abac`:

After	Interval
a	$[0, 0.6)$
b	$[0.36, 0.54)$
a	$[0.36, 0.468)$
c	$[0.4572, 0.468)$

The width is $0.6 \times 0.3 \times 0.6 \times 0.1 = 0.0108$, or $-\log_2 0.0108 \approx 6.5$ bits of information. The 7-bit binary fraction $0.0111011_2 = 0.4609375$ falls inside the interval, so 7 bits identify the message (plus a way to signal its end, such as a length or an end-of-message symbol).

Real coders use integer arithmetic with renormalization so that the interval never underflows, and they usually use **adaptive** probability models that are updated as symbols arrive, so no table needs to be sent. JPEG 2000's coefficient coder feeds an adaptive binary arithmetic coder [14], and the learned codecs in the Modern view section also end with arithmetic coding.

LZW coding

In plain words: build a dictionary of phrases as you go; whenever a phrase repeats, send its dictionary index instead of spelling it out.

The Lempel–Ziv–Welch (LZW) algorithm [12] starts with a dictionary holding all single symbols (codes 0–255 for 8-bit pixels). It reads the input, growing the current phrase w while w followed by the next symbol is still in the dictionary. When it is not, the coder outputs the code for w, adds

the extended phrase as a new entry (256, 257, ...), and restarts from the new symbol. The decoder rebuilds the same dictionary from the codes it receives, so no dictionary is transmitted.

For a row that alternates `39 39 126 126` four times (16 pixels), the output is ten codes:

```
def lzw_encode(seq):
    table = {(v,): v for v in range(256)}
    w, out = (), []
    for x in seq:
        wx = w + (x,)
        if wx in table:
            w = wx
        else:
            out.append(table[w])
            table[wx] = len(table)
            w = (x,)
    if w:
        out.append(table[w])
    return out

row = [39, 39, 126, 126] * 4
print(lzw_encode(row))   # [39, 39, 126, 126, 256, 258, 260, 259, 257, 126]
```

Codes 256 onward stand for `39 39`, `39 126`, `126 126`, `126 39`, `39 39 126`, and so on. The longer the repetition, the longer the phrases and the better the ratio. LZW needs no prior knowledge of the probabilities, which made it popular in general-purpose file formats. PNG uses the related LZ77-style deflate method [4].

Run-length coding

In plain words: “0000 11 00000 1” becomes “four 0s, two 1s, five 0s, one 1”.

Run-length encoding (RLE) replaces each run of identical values with a pair (value, length). For binary images only the lengths are needed, if the first run’s color is agreed on. The run lengths themselves are then coded with a variable-length code, since short runs are more frequent than long ones.

```
import numpy as np

def rle(row):
    starts = np.r_[0, np.flatnonzero(np.diff(row)) + 1]
    lengths = np.diff(np.r_[starts, len(row)])
    return list(zip(row[starts].tolist(), lengths.tolist()))

print(rle(np.array([0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1])))  # [(0, 4), (1, 2), (0, 5), (1,
 ↪  1)]
```

RLE works well for documents, line drawings and bit planes, and poorly for noisy photographs where runs are short. Two-dimensional variants code the positions of color transitions in the current row relative to the transitions in the row above, which exploits vertical correlation too. JPEG uses a form of run-length coding for the many zero-valued coefficients produced by quantization [13].

Symbol-based coding

In plain words: a scanned page is mostly the same few letters again and again. Store each letter shape once, then just say "letter 7 goes here".

In **symbol-based** (token-based) coding, frequently occurring sub-images, called **symbols**, are stored once in a **symbol dictionary**. The image becomes a list of triplets (x_i, y_i, t_i): the position of each occurrence and the index t_i of its dictionary entry. A text page with thousands of characters but about a hundred distinct glyph shapes compresses very well, since each occurrence costs two coordinates and an index, which are then entropy-coded.

If an occurrence may match an entry that is only *similar*, the method becomes lossy. That improves the ratio, but a careless matcher can substitute a wrong character, such as two digits that look alike in a low-resolution scan. Safe systems keep the match threshold conservative, or also code the residual between occurrence and token to stay lossless.

Bit-plane coding

In plain words: split an 8-bit image into eight black-and-white pictures, one per bit, and compress each with a binary method.

A gray level $a = a_7 2^7 + a_6 2^6 + \cdots + a_0 2^0$ has eight bits $a_7, \ldots, a_0$. Collecting bit k of every pixel gives **bit plane** k (Figure 8.3). The high-order planes look like coarse versions of the picture and contain large uniform areas; the low-order planes look like noise. Each plane can be coded with run-length or other binary methods.

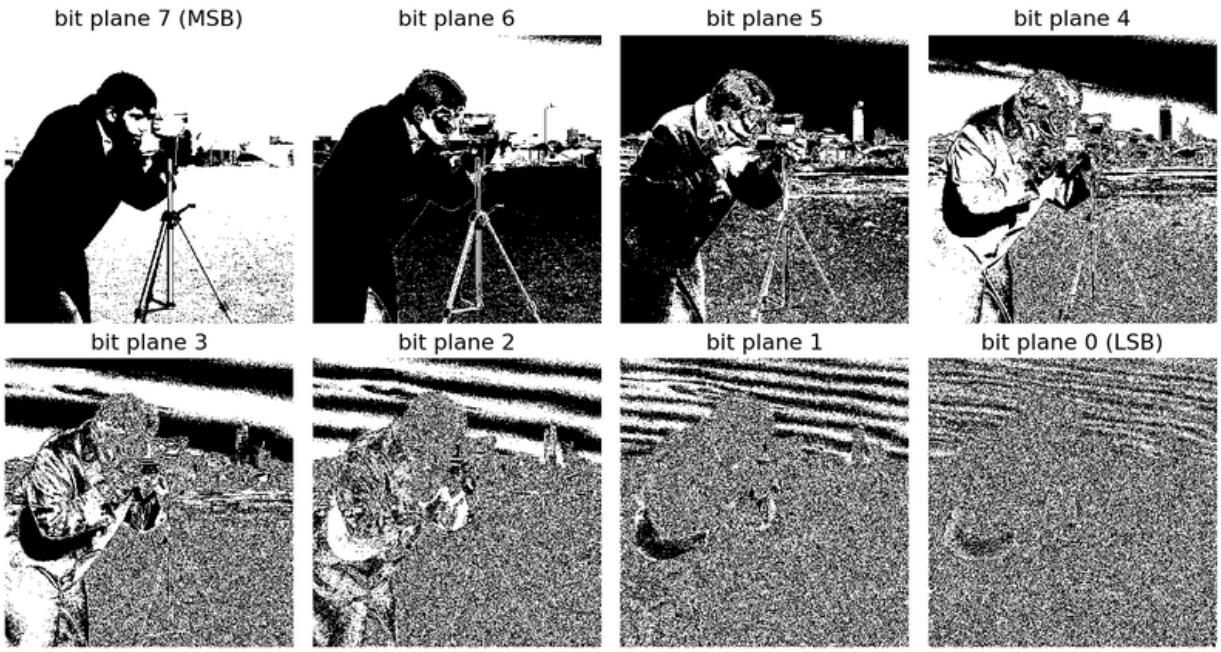


Figure 8.3 — Bit planes of an 8-bit image. Structure lives in the top planes; the bottom planes are close to random.

A weakness of the plain binary code is that a small change in gray level can flip many bits: 127 is `01111111` and 128 is `10000000`. The **Gray code** fixes this by making consecutive values differ in exactly one bit:

$$g_7 = a_7, \qquad g_i = a_i \oplus a_{i+1}, \quad 0 \le i \le 6,$$

where $\oplus$ is exclusive OR. With Gray-coded planes, smooth areas produce longer runs. On the camera image, bit plane 6 changes value between horizontal neighbors 6.4% of the time; its Gray-coded counterpart changes only 3.2% of the time, so its runs are about twice as long on average. In NumPy the conversion is one line: `gray = img ^ (img >> 1)`.

JPEG 2000 also codes its wavelet coefficients bit plane by bit plane, from the most significant down, which is what makes its bit stream truncatable at any rate [14].

Block transform coding

In plain words: cut the image into small tiles, describe each tile as a mix of standard patterns (smooth to busy), keep the important patterns, and throw away the fine ones the eye barely notices.

A **block transform coder** splits the image into non-overlapping $n \times n$ subimages, transforms each one, quantizes the coefficients, and entropy-codes the result. The transform does not compress by itself; it **packs** most of the energy into a few coefficients so that the rest can be quantized coarsely or dropped.

Transform selection

For an $n \times n$ block $g(x, y)$, the 2-D **discrete cosine transform** (DCT) is

$$T(u,v) = \alpha(u)\alpha(v) \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} g(x,y) \cos \frac{(2x+1)u\pi}{2n} \cos \frac{(2y+1)v\pi}{2n},$$

with $\alpha(0) = \sqrt{1/n}$ and $\alpha(u) = \sqrt{2/n}$ for $u = 1, \ldots, n-1$; u, v index the horizontal and vertical frequencies. The Karhunen–Loève transform (KLT, Chapter 7) packs energy best in the mean-square sense, but its basis depends on the data and must be computed and sent. The DCT is fixed, has fast algorithms, and for highly correlated images its basis is close to the KLT's. It also beats the DFT for blocks: the DFT treats a block as periodic, so the jump between its left and right edges creates spurious high frequencies, while the DCT's implied even symmetric extension has no such jump.

Subimage size

Larger blocks capture more correlation and pack energy better, but cost more computation, and an error spreads over a larger area. Gains level off quickly beyond about 8×8 or 16×16. JPEG chose a fixed 8×8; many newer codecs choose among several block sizes, using large blocks in smooth regions and small ones around detail.

Bit allocation

After the transform, we decide how many bits each coefficient gets.

- **Zonal coding** keeps a fixed zone of coefficients (usually the low frequencies, which have the largest variance across blocks) in every block.
- **Threshold coding** keeps whichever coefficients exceed a threshold in each block, so the kept set adapts to content.

The common practical form of threshold coding divides each coefficient by an entry of a **normalization (quantization) array** Z and rounds:

$$\hat{T}(u,v) = \text{round}\left[\frac{T(u,v)}{Z(u,v)}\right],$$

where $\hat{T}(u,v)$ is the integer that is actually coded. The decoder multiplies back, $\dot{T}(u,v) = \hat{T}(u,v)\, Z(u,v)$, and inverts the DCT. Large entries of Z at high frequencies throw away fine detail; small entries at low frequencies keep the coarse structure.

JPEG baseline

The JPEG baseline coder for 8-bit grayscale [2][13] does exactly this:

1. Shift pixel values by -128 so they are centered on zero.
2. Compute the 8×8 DCT of each block.
3. Quantize with a Z table. The standard gives an example luminance table; a widely used convention scales it by a quality factor $q \in [1, 100]$, with scale $s = 5000/q$ for $q < 50$ and $s = 200 - 2q$ otherwise (in percent).
4. Code the DC coefficient $\hat{T}(0,0)$ as the difference from the previous block's DC.
5. Reorder the 63 AC coefficients in a **zigzag** from low to high frequency, so zeros cluster at the end, and code them as (run of zeros, nonzero value) pairs with Huffman tables, ending each block with an end-of-block symbol.

Figure 8.4 follows one dark, smooth block through steps 2–3 at $q = 50$. It has a large DC term (-603, because of the level shift) and a few low-frequency terms. After quantization only 6 of 64 integers are nonzero, yet the reconstruction is off by only a few gray levels.

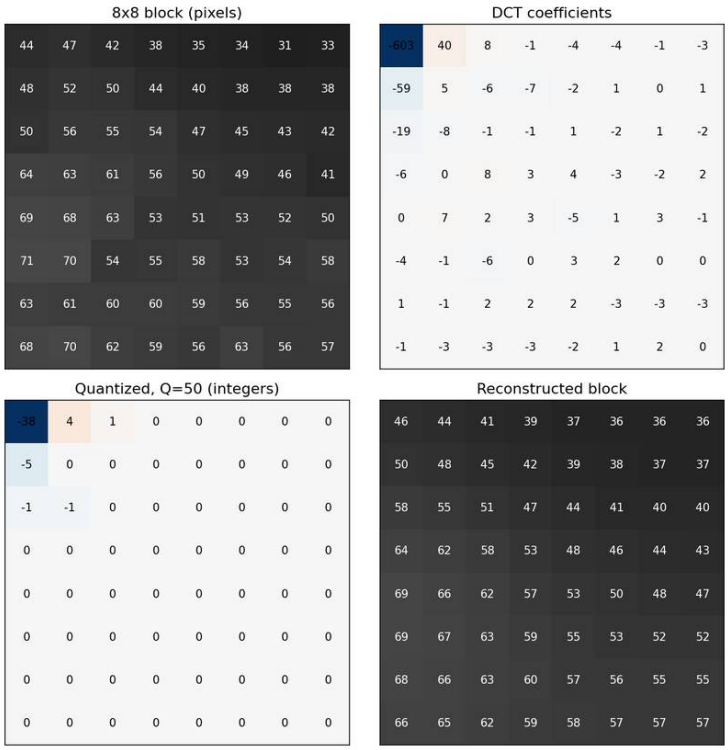


Figure 8.4 — One 8×8 block through the JPEG pipeline at quality 50. Quantization zeroes 58 of the 64 coefficients.

Figure 8.5 applies the same pipeline (without the entropy coder) to the whole image at four quality settings. PSNR drops from 40.3 dB at $q = 90$ to 26.3 dB at $q = 5$, while the fraction of nonzero coefficients falls from 31.3% to 2.4%. At low quality the familiar artifacts appear: **blocking** (visible 8×8 seams, because each block is coded independently) and **ringing** near edges (because the high frequencies that make a sharp edge were removed).



Figure 8.5 — JPEG-style DCT quantization at several quality settings. Blocking becomes obvious at $q = 5$.

Interactive demonstration (online): open the online edition of this tutorial to use this demonstration.Open demonstration

```
import numpy as np
from scipy.fft import dctn, idctn

Q50 = np.array([[16,11,10,16,24,40,51,61],[12,12,14,19,26,58,60,55],
                [14,13,16,24,40,57,69,56],[14,17,22,29,51,87,80,62],
                [18,22,37,56,68,109,103,77],[24,35,55,64,81,104,113,92],
                [49,64,78,87,103,121,120,101],[72,92,95,98,112,100,103,99]])

def dct_codec(img, Q):
    f = img.astype(float) - 128                       # level shift
    H, W = f.shape
    B = f.reshape(H//8, 8, W//8, 8).swapaxes(1, 2)   # (H/8, W/8, 8, 8) blocks
    q = np.round(dctn(B, axes=(2, 3), norm="ortho") / Q)
    rec = idctn(q * Q, axes=(2, 3), norm="ortho")
    rec = rec.swapaxes(1, 2).reshape(H, W) + 128
    return np.clip(rec.round(), 0, 255).astype(np.uint8), q

rec, q = dct_codec(img, Q50)     # PSNR 32.6 dB, 12.0% nonzero coefficients
```

Drag the slider below to compare a lossless PNG of a color test image (465 kB) with a real JPEG at quality 4 (7.6 kB, about 60 times smaller). Look at the flat background, the face and the color edges: blocks, ringing and color bleeding (the color channels are stored at lower resolution and quantized harder) are all visible.



Original (*PNG*)



JPEG q=4

Astronaut test image, lossless versus heavily quantized JPEG

Predictive coding

In plain words: guess each pixel from the ones you have already sent, and only send how wrong the guess was. Good guesses mean small numbers, and small numbers are cheap.

Lossless predictive coding

The encoder forms a prediction $\hat{f}(n)$ of pixel $f(n)$ from previously coded pixels and codes the **prediction error**

$$e(n) = f(n) - \hat{f}(n), \qquad \hat{f}(n) = \operatorname{round}\left[\sum_{i=1}^{m} \alpha_i f(n-i)\right],$$

where m is the predictor order and α_i are the prediction coefficients. The decoder computes the same prediction and adds $e(n)$ back, so the reconstruction is exact. Figure 8.1 already showed the payoff: the simplest predictor, "same as the left neighbor", lowers the entropy from 7.23 to 4.70 bits/pixel.

PNG lets each row choose among five filters: none, left (Sub), above (Up), the average of left and above, and the Paeth predictor [4]. JPEG-LS [10] uses the **median edge detector**: with a the left, b the upper and c the upper-left neighbor,

$$\hat{f} = \begin{cases} \min(a,b) & c \geq \max(a,b) \\ \max(a,b) & c \leq \min(a,b) \\ a + b - c & \text{otherwise.} \end{cases}$$

The first two cases pick the neighbor on the correct side of a horizontal or vertical edge; the third assumes a smooth plane.

Motion compensation

Video adds a time axis. Instead of predicting from neighbors in the same frame, a **motion-compensated** predictor copies a block from a previously decoded frame, displaced by a **motion vector** (d_x, d_y) found by searching for the best match (for example, the smallest sum of absolute differences). The encoder sends the motion vector and the residual, usually transform-coded as in the previous section. Frames coded only from themselves (intra frames) provide entry points for random access; predicted frames are much smaller. HEVC, whose still-image form is stored in HEIC files [7], and AV1, used by AVIF [6], both combine this kind of inter-frame prediction with transform coding.

Lossy predictive coding (DPCM)

If we quantize the prediction error, $\dot{e}(n) = Q[e(n)]$, we get **differential pulse-code modulation** (DPCM). The key detail is that the encoder must predict from the *reconstructed* values $\dot{f}(n) = \dot{e}(n) + \hat{f}(n)$, the same values the decoder will have. If the encoder predicted from the originals instead, quantization errors would accumulate in the decoder (drift).

```
import numpy as np

def dpcm(row, step):
    """Lossy DPCM with previous-sample prediction and a uniform quantizer."""
    rec = np.empty(len(row)); prev = 128.0; idx = []
    for i, x in enumerate(row.astype(float)):
        e = x - prev                        # prediction error
        k = int(np.round(e / step))         # quantizer index: this is what is sent
        prev = prev + k * step              # reconstruction, identical in the decoder
        rec[i] = prev; idx.append(k)
    return np.array(idx), rec
```

On one row of the camera image with step 8, the indices have an entropy of 1.87 bits/sample and the reconstruction has a PSNR of 40.8 dB. With a very coarse quantizer, two typical distortions appear: **slope overload** (the reconstruction cannot climb fast enough at a steep edge) and **granular noise** (it jitters in flat areas).

Online only: Optimal predictors and quantizers — https://blog.shyandram.dev/en/learn/image-processing/compression-watermarking/notes/optimal-predictors-and-quantizers/

Wavelet coding

In plain words: instead of chopping the image into tiles, split it into a blurry small version plus layers of detail at different scales. Most detail values are near zero, so they are cheap to store, and there are no tile seams.

A wavelet coder applies the discrete wavelet transform of Chapter 7, quantizes the coefficients, and entropy-codes them. Because the basis functions overlap and are localized at many scales, a wavelet coder has no block boundaries and degrades by blurring and mild ringing rather than by blocking. Figure 8.6 compares an 8×8 DCT and a five-level wavelet transform that each keep the same 2% of their coefficients (the largest in magnitude). The DCT result is a mosaic at 26.8 dB; the wavelet result is smoother and reaches 28.6 dB.



Figure 8.6 — Same coefficient budget (2%), different transforms. Block DCT produces seams; the wavelet transform does not.

```
import numpy as np, pywt

def wavelet_code(img, keep=0.02, wavelet="bior4.4", level=5):
    c = pywt.wavedec2(img.astype(float), wavelet, level=level, mode="periodization")
    arr, slices = pywt.coeffs_to_array(c)
    t = np.quantile(np.abs(arr), 1 - keep)            # keep the largest 2%
    arr[np.abs(arr) < t] = 0
    c = pywt.array_to_coeffs(arr, slices, output_format="wavedec2")
    return np.clip(pywt.waverec2(c, wavelet, mode="periodization"), 0, 255)
```

This toy counts coefficients, not bits. A real coder must also signal *which* coefficients survived, which is where clever significance coding earns its keep.

Wavelet selection

The wavelet determines how well energy is packed and what artifacts look like. Image coders favor **biorthogonal** wavelets because they can be symmetric (so edges are not shifted) and have short filters. JPEG 2000 specifies two: an irreversible 9/7-tap filter pair for lossy coding and a reversible integer 5/3 filter pair for lossless coding [14]. PyWavelets' `bior4.4` used above has the same 9- and 7-tap lengths.

Decomposition level

Each level splits the current approximation into four half-size subbands. More levels make the representation sparser, but gains shrink after about five levels for typical image sizes, and quantizing very coarse coefficients affects large areas.

Quantizer design

Wavelet coders usually use a uniform scalar quantizer with an enlarged **dead zone** around zero:

$$q_b = \operatorname{sign}(c_b) \left\lfloor \frac{|c_b|}{\Delta_b} \right\rfloor,$$

where c_b is a coefficient in subband b, Δ_b is that subband's step size and q_b the integer index. Small coefficients fall into the zero bin; they can encode genuine texture or faint detail as well as noise, so discarding them is a rate–distortion choice. Step sizes are chosen per subband, coarser for the high-frequency subbands where errors are less visible.

JPEG 2000

JPEG 2000 [3][14] builds a full system around these pieces. The image can be split into large tiles, color components are decorrelated with a component transform, and each tile is wavelet transformed and dead-zone quantized. Each subband is divided into small **code blocks** that are coded independently bit plane by bit plane with an adaptive arithmetic coder. Finally, the coded pieces are arranged into **quality layers**, so a single file can be decoded at lower resolution or lower quality just by reading a prefix of the stream. It also supports lossless coding and regions of interest coded at higher quality.

Digital image watermarking

In plain words: a watermark is extra information hidden in an image, like a signature, an owner ID or a "this was generated by a model" flag, that travels with the picture.

Kinds of watermarks

- **Visible** watermarks are overlays (a semi-transparent logo); they degrade the image and can often be cropped or painted out. **Invisible** watermarks stay below the visibility threshold and are recovered by a detector.
- A **robust** watermark survives compression, resizing, filtering and cropping (for ownership and provenance). A **fragile** one breaks under any change (for authentication). **Semi-fragile** marks survive mild compression but not content edits.
- **Private** (non-blind) detection needs the original image; **blind** detection does not.

Every scheme trades off **imperceptibility** (how little the image changes), **robustness**, **capacity** (how many bits are hidden) and **security** (whether someone without the key can find, remove or forge the mark).

A fragile example: least significant bits

The simplest invisible watermark replaces the least significant bit (LSB) of each pixel with a bit of the mark: $f_w = 2\lfloor f/2 \rfloor + w$, with $w \in \{0, 1\}$. Each changed 8-bit pixel differs by exactly one gray level. If a fraction q of pixels changes, the MSE is q and the PSNR is $10 \log_{10}(255^2/q)$ dB. Thus about half the pixels changing gives roughly 51.14 dB; the worst case $q = 1$ gives 48.13 dB. If $q = 0$, the images are identical and PSNR is infinite. These values follow directly from the PSNR definition above, not from an empirical guarantee for every watermark.

```
def lsb_embed(cover, mark):          # cover: uint8 image, mark: 0/1 array of the same shape
    return (cover & 0xFE) | mark

def lsb_extract(img):
    return img & 1
```

Figure 8.7 embeds a binary "DIP" logo. Extraction is perfect from the PNG. After a single JPEG compression at quality 95, which looks identical to the eye, only 55% of the extracted bits match the mark, barely better than a coin flip. The LSB plane is the noise-like part of the image that every lossy coder discards first. This example demonstrates fragility, not authenticated integrity: it contains no keyed check binding the mark to the image, and anyone can replace its bits. JPEG damage can be detected by comparing with the expected mark, but successful extraction alone does not prove that the remaining image content is unchanged.

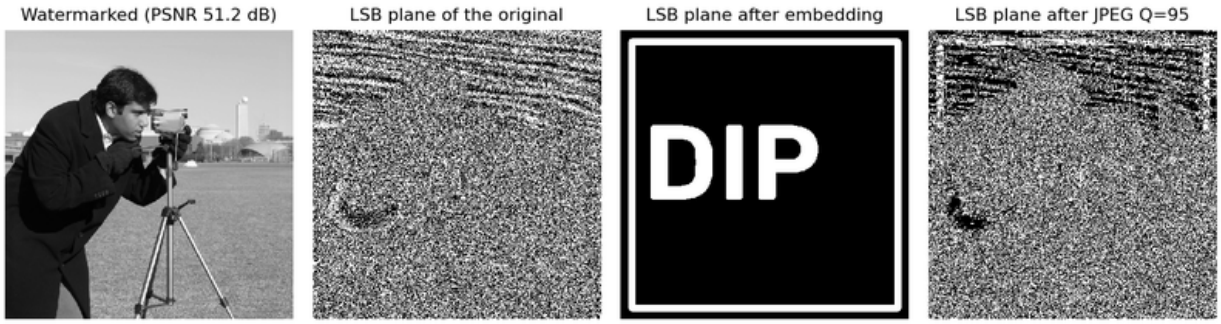


Figure 8.7 — An LSB watermark is invisible (51 dB) and perfectly recoverable, until a mild JPEG pass wipes it out.

A robust example: DCT-domain watermark

To survive JPEG, hide the mark where JPEG keeps information: in low- to mid-frequency DCT coefficients. The scheme below is a simple one of our own. In each 8×8 block it compares two mid-frequency coefficients, $C(2, 3)$ and $C(3, 2)$, which JPEG quantizes with similar step sizes. To embed bit 1, it makes $C(2, 3)$ exceed $C(3, 2)$ by at least a margin α; to embed bit 0, the reverse. The detector only checks which coefficient is larger, so it is blind.

```
import numpy as np
from scipy.fft import dctn, idctn

def blocks(img):
    H, W = img.shape
    return img.astype(float).reshape(H//8, 8, W//8, 8).swapaxes(1, 2).reshape(-1, 8, 8)

def unblocks(B, H, W):
    return B.reshape(H//8, W//8, 8, 8).swapaxes(1, 2).reshape(H, W)

def dct_embed(img, bits, alpha=12.0):
    C = dctn(blocks(img), axes=(1, 2), norm="ortho")
    for i, b in enumerate(bits):               # one bit per 8x8 block
        x, y = C[i, 2, 3], C[i, 3, 2]
        if (x - y if b else y - x) < alpha:    # enforce the order with margin alpha
            m = (x + y) / 2
            s = alpha / 2 if b else -alpha / 2
            C[i, 2, 3], C[i, 3, 2] = m + s, m - s
    out = unblocks(idctn(C, axes=(1, 2), norm="ortho"), *img.shape)
    return np.clip(out.round(), 0, 255).astype(np.uint8)

def dct_extract(img):
    C = dctn(blocks(img), axes=(1, 2), norm="ortho")
    return (C[:, 2, 3] > C[:, 3, 2]).astype(int)
```

On the 512×512 camera image this hides 4,096 bits (one per block). The margin α is the robustness dial:

Margin α	PSNR of marked image	Bit errors after JPEG q=90	q=75	q=50
12	43.2 dB	0.10%	0.56%	36.6%
30	38.4 dB	0%	0%	0.05%

A small margin is nearly invisible but breaks once the JPEG step size at those positions exceeds the margin. A larger margin survives quality 50 at the cost of about 5 dB PSNR. Real systems add error-correcting codes, spread each bit over many blocks with a secret key, and adapt the strength to local texture, since the eye tolerates larger changes in busy regions.

Modern view

The classical pipeline (transform, quantize, entropy code) still underlies every deployed image codec. What has changed since about 2016 is that each stage can be *learned*, and that the same idea of trained encoders and decoders has reshaped watermarking.

Surveys worth reading

- **Wallace (1992)** [13]: the clearest overview of JPEG's design.

- **Skodras, Christopoulos and Ebrahimi (2001)** [14]: a tutorial review of JPEG 2000, whose value lies more in its features (scalability, region-of-interest coding, lossless operation) than in raw efficiency.
- **Hu, Yang, Ma and Liu (2021)** [16]: a survey of end-to-end learned lossy compression along network architecture, entropy model and rate control.
- **Yang, Mandt and Theis (2023)** [17]: a monograph-length introduction to neural compression for a machine-learning audience.
- **Zhong, Das, Alrasheedi and Tanvir (2023)** [18]: a survey of deep-learning image watermarking, sorted into three families.

Online only: The Modern view surveys in more detail — https://blog.shyandram.dev/en/learn/image-processing/compression-watermarking/papers/the-modern-view-surveys-in-more-detail/

Learned image compression

Ballé et al. [15] replaced the DCT with a learned nonlinear **analysis transform** (a convolutional network) that maps the image x to a latent y, which is quantized to $\hat{y}$ and decoded by a learned **synthesis transform**. The whole system is trained to minimize a rate-distortion loss

$$\mathcal{L} = R + \lambda D = E\left[-\log_2 p(\hat{y})\right] + \lambda\, E\left[d(x, \hat{x})\right],$$

where R is the expected code length under the entropy model p, D is a distortion such as MSE, and λ sets the trade-off. Their key contribution was the **scale hyperprior**: a second, small latent z is sent as side information and predicts the scale of each element of y, so the entropy model adapts to each image. That is the same idea as JPEG's per-image Huffman tables or JPEG-LS's context modeling, but learned end to end. During training, rounding is replaced by additive uniform noise so gradients can flow.

What did not change: quantization is still rounding, and the bits still come from an arithmetic coder [11]. Learned codecs are better mappers and better probability models inside the same encoder diagram as before. Their practical obstacles are decoder complexity and the need for bit-exact, platform-independent entropy models.

Modern standard codecs

Meanwhile, standardized codecs kept improving with hand-designed tools. AVIF [6] and HEIC [7] reuse the intra-frame coding tools of the AV1 and HEVC video codecs, JPEG XL [5] targets photography and the web, and PNG [4] remains the default for lossless graphics. None of these replaced the textbook's ideas; they combine predictive coding, block transforms and adaptive entropy coding more flexibly.

Online only: Modern standard codecs in more detail — https://blog.shyandram.dev/en/learn/image-processing/compression-watermarking/notes/modern-standard-codecs-in-more-detail/

Deep-learning watermarking

HiDDeN [19] jointly trains an encoder that hides a bit string in a cover image, a decoder that recovers it, and a **noise layer** between them that simulates cropping, blur and JPEG (through differentiable approximations, since real JPEG has no gradient). An adversarial loss keeps the marked images natural. Robustness is learned from simulated attacks rather than designed in by choosing coefficients as we did above, so it is only as good as those simulations.

Generative models opened a new use: marking images *at creation*. Stable Signature [20] fine-tunes the decoder of a latent diffusion model so every generated image carries a fixed hidden signature, recoverable by an extractor even after cropping and other edits, with a false-positive rate below one in a million. This answers a provenance question (was this image made by model X?) rather than an ownership one. Removal and forgery attacks remain open problems.

Key takeaways

- Compression removes coding redundancy, spatial/temporal redundancy and irrelevant information; only the last one is lossy.
- Entropy is the lower bound for lossless coding of independent symbols, and the choice of symbols matters: prediction differences have far lower entropy than raw pixels.
- Huffman codes are optimal per symbol; arithmetic coding approaches the entropy of the whole message; Golomb, LZW, run-length, symbol-based and bit-plane coding exploit particular structure.
- JPEG = level shift + 8×8 DCT + quantization table + zigzag + run-length/Huffman coding; quality is set by scaling the table, and low quality shows blocking and ringing.
- Predictive coders must predict from reconstructed values; motion-compensated prediction is the heart of video coding.
- Wavelet coding (JPEG 2000) avoids block seams and supports scalable, lossless and region-of-interest coding.
- LSB watermarks are invisible but fragile; DCT-domain marks survive JPEG when their strength exceeds the quantization step.
- Learned codecs and deep watermarks replace hand-designed transforms with trained networks but keep quantization and arithmetic coding at their core.

Exercises

1. A source has four symbols with probabilities 0.5, 0.25, 0.125 and 0.125. Build a Huffman code, compute its average length and the entropy, and explain why they are equal here.

   Hint

   Codewords `0`, `10`, `110`, `111`; both values are 1.75 bits. All probabilities are powers of $1/2$, so $-\log_2 P(a_j)$ is an integer and each codeword can have exactly that length.

2. Prediction residuals $e = 0, -1, 2, -3, 1$ are mapped with $M(e)$ from the Golomb section and coded with G_2. Write the bit string and count its bits. Would G_4 be shorter?

Hint

$M(e) = 0, 1, 4, 5, 2$. With $m = 2$ the codes are `00`, `01`, `1100`, `1101`, `100` (15 bits). With $m = 4$ they are `00 0`, `001`, `1000`, `1001`, `010` (17 bits). For small residuals the smaller parameter wins.

3. Using the arithmetic coding example ($P(a) = 0.6$, $P(b) = 0.3$, $P(c) = 0.1$), decode the first three symbols of the tag 0.70.

Hint

0.70 lies in $[0.6, 0.9)$, so the first symbol is `b`. Rescale: $(0.70 - 0.6)/0.3 = 0.333$, which lies in a's range. Rescale again: $0.333/0.6 = 0.556$, again `a`. The message starts `baa`.

4. Modify the DPCM snippet so the encoder predicts from the *original* previous pixel instead of the reconstructed one, but keep the decoder unchanged. Measure the PSNR along a row and explain what you see.

Hint

The decoder adds quantized errors to its own reconstruction, which no longer matches the encoder's reference. Errors accumulate along the row (drift). On row 200 of the camera image with step 8, PSNR falls from 40.8 dB to about 17 dB.

5. Change the LSB watermark to use bit plane 3 instead of bit plane 0. Measure the PSNR of the marked image and the bit agreement after JPEG at quality 95 and 75. What did you gain and what did you pay?

Hint

Each pixel can now change by up to 8 gray levels, so PSNR falls by roughly $20 \log_{10} 8 \approx 18$ dB (about 33 dB with a random mark). Bit agreement after JPEG rises to about 82% at quality 95 but drops to about 57% at quality 75: you paid a lot of visibility for little robustness, compared with the DCT-domain scheme.

6. Explain why a wavelet coder at very low bit rates looks blurry while a block DCT coder looks blocky, in terms of the support of the basis functions that survive quantization.

Hint

The surviving DCT basis functions are confined to their 8×8 block, so neighboring blocks are approximated independently and meet at seams. The surviving coarse wavelet functions are smooth and overlap their neighbors, so the approximation is continuous but lacks fine detail.

References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018, Ch. 8. publisher page
2. Joint Photographic Experts Group, "JPEG 1 (ISO/IEC 10918-1 | ITU-T Rec. T.81): Digital compression and coding of continuous-tone still images," JPEG overview page. jpeg.org
3. Joint Photographic Experts Group, "JPEG 2000 (ISO/IEC 15444)," JPEG overview page. jpeg.org
4. W3C, "Portable Network Graphics (PNG) Specification (Third Edition)," W3C Recommendation, 2025. w3.org
5. Joint Photographic Experts Group, "JPEG XL (ISO/IEC 18181)," JPEG overview page. jpeg.org
6. Alliance for Open Media, "AV1 Image File Format (AVIF)," v1.2.0, 2025. aomedia-codec.github.io
7. Nokia Technologies, "High Efficiency Image File Format (HEIF), ISO/IEC 23008-12: technical information." nokiatech.github.io
8. D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, pp. 1098–1101, 1952. doi
9. S. W. Golomb, "Run-length encodings," *IEEE Transactions on Information Theory*, vol. 12, pp. 399–401, 1966. doi
10. M. J. Weinberger, G. Seroussi and G. Sapiro, "The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS," *IEEE Transactions on Image Processing*, vol. 9, pp. 1309–1324, 2000. doi
11. I. H. Witten, R. M. Neal and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, pp. 520–540, 1987. doi
12. T. A. Welch, "A Technique for High-Performance Data Compression," *Computer*, vol. 17, pp. 8–19, 1984. doi
13. G. K. Wallace, "The JPEG still picture compression standard," *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992. doi
14. A. Skodras, C. Christopoulos and T. Ebrahimi, "The JPEG 2000 still image compression standard," *IEEE Signal Processing Magazine*, vol. 18, pp. 36–58, 2001. doi
15. J. Ballé, D. Minnen, S. Singh, S. J. Hwang and N. Johnston, "Variational image compression with a scale hyperprior," *ICLR*, 2018. arXiv
16. Y. Hu, W. Yang, Z. Ma and J. Liu, "Learning End-to-End Lossy Image Compression: A Benchmark," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021. arXiv
17. Y. Yang, S. Mandt and L. Theis, "An Introduction to Neural Data Compression," *Foundations and Trends in Computer Graphics and Vision*, vol. 15, no. 2, 2023. arXiv
18. X. Zhong, A. Das, F. Alrasheedi and A. Tanvir, "A Brief Yet In-Depth Survey of Deep Learning-Based Image Watermarking," *Applied Sciences*, 2023. arXiv
19. J. Zhu, R. Kaplan, J. Johnson and L. Fei-Fei, "HiDDeN: Hiding Data With Deep Networks," *arXiv:1807.09937*, 2018. arXiv
20. P. Fernandez, G. Couairon, H. Jégou, M. Douze and T. Furon, "The Stable Signature: Rooting Watermarks in Latent Diffusion Models," *ICCV*, 2023. arXiv

IP 09 • Morphological Image Processing

Probe shapes with small structuring elements: erosion, dilation, opening, closing, hit-or-miss, skeletons, reconstruction and grayscale morphology, with scikit-image code.

Online edition: https://blog.shyandram.dev/en/learn/image-processing/morphology/

What you'll learn

- How to treat a binary image as a set of pixel coordinates, and how a small **structuring element** probes it
- The two primitive operations, **erosion** and **dilation**, and how they combine into **opening** and **closing**
- How the **hit-or-miss transform** finds exact pixel patterns such as corners and isolated points
- Classic algorithms built from those primitives: boundaries, hole filling, connected components, convex hulls, thinning, skeletons and pruning
- **Morphological reconstruction**, which removes objects without distorting the ones that survive
- **Grayscale morphology**: gradients, top-hat correction of uneven lighting, and size distributions (granulometry)

The big picture

Most of the filters you met in earlier chapters were *linear*: each output pixel is a weighted sum of input pixels. Morphology is different. It asks yes-or-no questions about **shape**: "Does this little shape fit here?" "Does it touch anything here?" The answers are built only from minimum, maximum, intersection and union. No multiplication is involved.

> **Explain like I'm five**
>
> Imagine a sheet of cookie dough with a funny outline, and a small round cookie cutter. Slide the cutter everywhere over the table. **Erosion** marks every spot where the cutter fits *completely* on the dough, so the dough shrinks and thin bits vanish. **Dilation** marks every spot where the cutter touches *any* dough, so the dough grows and small gaps fill in. **Opening** keeps only the dough you could actually cut cookies from; **closing** fills every dent that is too small for the cutter to poke into. Change the cutter's shape (a line, a square, a cross) and you ask a different question about the dough.

Why it matters: morphology is the standard toolbox for cleaning binary masks (from thresholding in IP 10, or from a neural network), for measuring objects (counting, sizing, finding their skeletons), and for correcting uneven backgrounds. Its theory, **mathematical morphology**, grew out of work by Georges Matheron and Jean Serra on random sets and on the geometry of porous materials [2], [3]. Haralick, Sternberg and Zhuang's tutorial brought it to the wider image-analysis

community [4], and Soille's book remains the standard practical reference [5]. This chapter follows the topic order of Gonzalez and Woods [1] and uses scikit-image [12] and SciPy for the code.

Preliminaries

In plain words: a binary image is a list of "on" pixel positions, and a structuring element is a tiny stencil with a marked centre that we slide over that list.

Images as sets

Let a binary image have foreground pixels (value 1) and background pixels (value 0). We describe the foreground by the set

$$A = \{(x, y) \in \mathbb{Z}^2 \mid I(x, y) = 1\},$$

where I is the image and (x, y) are integer pixel coordinates. The usual set operations now have image meanings: the **complement** A^c is the background, the **union** $A \cup C$ is a pixel-wise OR, the **intersection** $A \cap C$ is a pixel-wise AND, and the **difference** $A - C = A \cap C^c$ removes from A everything in C.

Structuring elements

A **structuring element** (SE) B is a second, small set of offsets with a designated **origin**, usually its centre. Think of it as the probe. Common choices are a 3×3 square, a cross (4-neighbourhood), a disk, or a line segment. The SE's shape decides which features the operation responds to: a disk is direction-neutral, a horizontal line responds only to horizontal structure. scikit-image calls an SE a **footprint**. In practice, an SE is stored as a small Boolean array whose `True` entries are its members; elements shown as "don't care" in diagrams are simply not in the set.

Reflection and translation

Two operations on sets appear in every definition below. The **reflection** of B is

$$\hat{B} = \{w \mid w = -b,\ b \in B\},$$

that is, every offset b is replaced by $-b$: the SE is rotated 180° about its origin. The **translation** of B by a vector $z = (z_1, z_2)$ is

$$(B)_z = \{c \mid c = b + z,\ b \in B\},$$

which places the SE's origin at pixel z. For symmetric SEs such as disks and squares, $\hat{B} = B$, so the reflection is invisible; for asymmetric ones it matters.

```
import numpy as np
from skimage import morphology

B = morphology.disk(2)            # 5x5 disk; origin at the centre (2, 2)
B_hat = B[::-1, ::-1]             # reflection: flip both axes
L = np.array([[1, 1, 0],
              [0, 1, 0],
              [0, 0, 0]], bool)   # an asymmetric element
print(L[::-1, ::-1].astype(int))  # its reflection points the other way
```

Erosion and dilation

In plain words: erosion keeps a pixel only if the whole stencil fits inside the shape there; dilation turns a pixel on if the stencil touches the shape at all.

Erosion

The **erosion** of A by B is

$$A \ominus B = \{z \mid (B)_z \subseteq A\},$$

the set of all positions z at which the translated SE lies entirely inside the foreground. Equivalently, $A \ominus B = \{z \mid (B)_z \cap A^c = \emptyset\}$: the SE must not overlap any background pixel. Erosion shrinks objects by roughly the SE's radius, deletes any part thinner than the SE, and separates objects joined by thin bridges. If B contains its origin, erosion is **anti-extensive**: $A \ominus B \subseteq A$.

Dilation

The **dilation** of A by B is

$$A \oplus B = \{z \mid (\hat{B})_z \cap A \neq \emptyset\},$$

the set of positions at which the reflected SE overlaps at least one foreground pixel. An equivalent form is the union of copies of B placed at every foreground pixel: $A \oplus B = \bigcup_{a \in A} (B)_a$. Dilation grows objects, bridges gaps narrower than the SE, and fills small holes. It is **extensive** ($A \subseteq A \oplus B$) when B contains its origin. Unlike erosion, dilation is commutative: $A \oplus B = B \oplus A$.

The reflection in the dilation formula is what makes dilation a true Minkowski sum and keeps it consistent with convolution's flip. For symmetric SEs it changes nothing.

Duality

Erosion and dilation are **duals** with respect to complement and reflection:

$$(A \ominus B)^c = A^c \oplus \hat{B}, \qquad (A \oplus B)^c = A^c \ominus \hat{B}.$$

Eroding the foreground is the same as dilating the background (with the reflected SE), then flipping the result. The proof is one line: $z \notin A \ominus B$ exactly when $(B)_z$ meets A^c, which is the definition of $z \in A^c \oplus \hat{B}$ (because $\hat{\hat{B}} = B$). In code, duality holds exactly only if you treat the image border consistently: pixels outside the image must count as foreground for the erosion when they count as background for the dilation of the complement.

```
import numpy as np
from scipy import ndimage as ndi
from skimage import data, morphology

A = ~data.horse()                # True = horse pixels
B = morphology.disk(3)

eroded  = ndi.binary_erosion(A, structure=B)
dilated = ndi.binary_dilation(A, structure=B)
print(A.sum(), eroded.sum(), dilated.sum())     # erosion shrinks, dilation grows

# Duality check. border_value=1 treats outside pixels as foreground for the
# erosion, which matches background-outside for the dilation of ~A.
lhs = ~ndi.binary_erosion(A, structure=B, border_value=1)
rhs = ndi.binary_dilation(~A, structure=B[::-1, ::-1])
print("duality holds:", np.array_equal(lhs, rhs))   # True
```

Opening and closing

In plain words: opening is "erode, then dilate back": it removes specks and thin parts but leaves big shapes nearly unchanged. Closing is "dilate, then erode back": it fills small holes and narrow cracks.

Definitions

The **opening** of A by B is erosion followed by dilation with the same SE,

$$A \circ B = (A \ominus B) \oplus B,$$

and the **closing** is dilation followed by erosion,

$$A \bullet B = (A \oplus B) \ominus B.$$

Erosion alone throws away detail but also shrinks everything; the second step restores the size of whatever survived. The pair is dual, just like erosion and dilation: $(A \bullet B)^c = A^c \circ \hat{B}$ and $(A \circ B)^c = A^c \bullet \hat{B}$. Closing the foreground is opening the background.

Geometric interpretation: the rolling ball

Opening has a direct picture. It equals the union of all translates of B that fit inside A:

$$A \circ B = \bigcup \{(B)_z \mid (B)_z \subseteq A\}.$$

Roll a ball (the SE) around *inside* the foreground, staying fully inside. Every pixel the ball can reach is kept; corners sharper than the ball, thin necks and specks smaller than the ball are lost. Closing is the same picture applied from the outside: roll the ball around in the *background*; every background pixel the ball cannot reach (narrow gaps, small holes, deep thin bays) becomes foreground.

Properties

Opening and closing are both **morphological filters** in the technical sense: they are increasing and idempotent [5].

- **Opening is anti-extensive:** $A \circ B \subseteq A$. It only removes pixels.
- **Closing is extensive:** $A \subseteq A \bullet B$. It only adds pixels.
- **Increasing:** if $C \subseteq D$ then $C \circ B \subseteq D \circ B$, and the same for closing. Bigger input never gives smaller output.
- **Idempotent:** $(A \circ B) \circ B = A \circ B$ and $(A \bullet B) \bullet B = A \bullet B$. Applying the same opening twice changes nothing, because whatever survived the first pass already consists of places where the ball fits.

Idempotence is the property that separates opening and closing from erosion and dilation: repeated erosions keep shrinking, while repeated openings stop. Figure 9.1 shows all four operations on a noisy silhouette. The opening removes the white specks in the background; the closing removes the black specks inside the horse; opening followed by closing removes both, a simple and very common **morphological noise filter**.

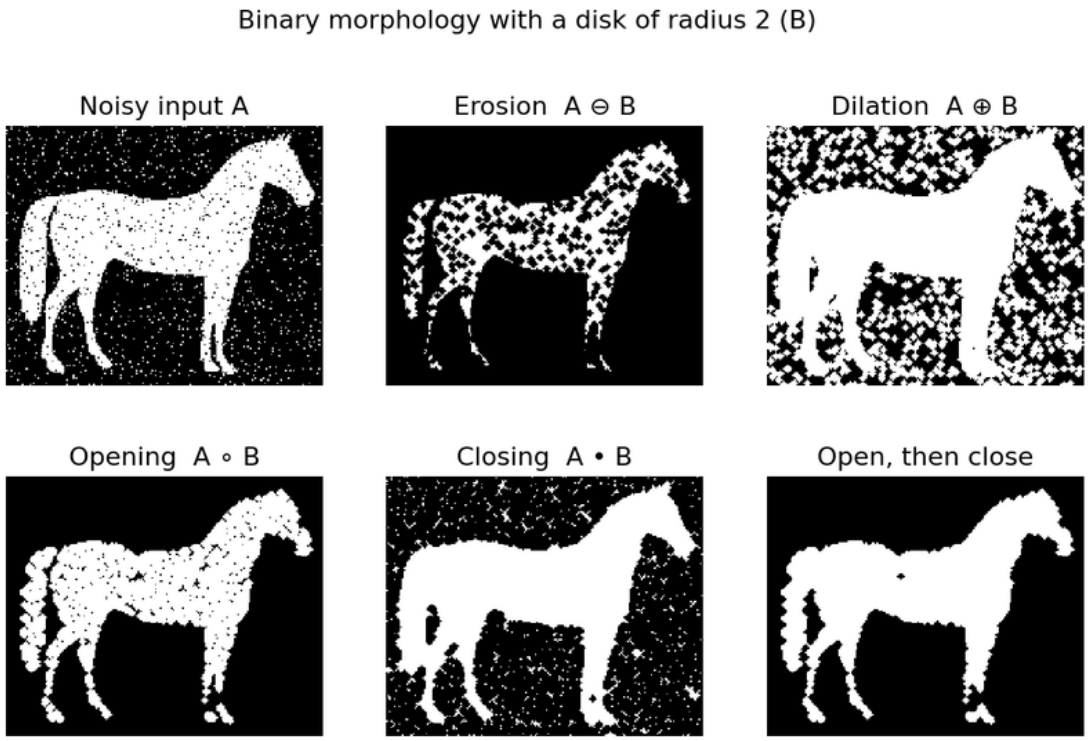


Figure 9.1 — Binary erosion, dilation, opening and closing of a horse silhouette with 4% of pixels flipped, using a disk of radius 2. Opening then closing removes both kinds of speck while keeping the outline.

Interactive demonstration (online): open the online edition of this tutorial to use this demonstration.Open demonstration

```
import numpy as np
from skimage import data, morphology

A = ~data.horse()
B = morphology.disk(4)
op = morphology.binary_opening(A, B)
cl = morphology.binary_closing(A, B)
print("opening is anti-extensive:", not (op & ~A).any())
print("closing is extensive:     ", not (A & ~cl).any())
print("opening is idempotent:    ", np.array_equal(morphology.binary_opening(op, B), op))
```

The hit-or-miss transform

In plain words: the hit-or-miss transform (HMT) is a template matcher for binary patterns. You say which pixels must be foreground and which must be background, and it lights up exactly where both conditions hold.

Use two disjoint SEs: B_1, the pixels that must lie in the foreground, and B_2, the pixels that must lie in the background. The HMT of A is

$$A \circledast B_{1,2} = \{z \mid (B_1)_z \subseteq A \text{ and } (B_2)_z \subseteq A^c\} = (A \ominus B_1) \cap (A^c \ominus B_2).$$

Here $B_{1,2}$ denotes the pair (B_1, B_2), $A \ominus B_1$ finds positions where the foreground pattern fits, and $A^c \ominus B_2$ finds positions where the background pattern fits. Pixels in neither B_1 nor B_2 are **don't-care** positions. It is common to draw the pair as one small grid with 1 (must be foreground), 0 (must be background) and blank (don't care).

Some useful templates:

- **Isolated point:** B_1 is the centre pixel; B_2 is its 8 neighbours.
- **Upper-left corner:** B_1 is the centre plus its right and lower neighbours; B_2 is the upper and left neighbours.
- **End point of a one-pixel line:** the centre and exactly one neighbour are foreground; the rest are background. You need one template per direction (8 rotations).

Figure 9.2 applies the first two templates. Note that the staircase outline of a digital disk contains several genuine upper-left corners at pixel level: the HMT is literal, not geometric.

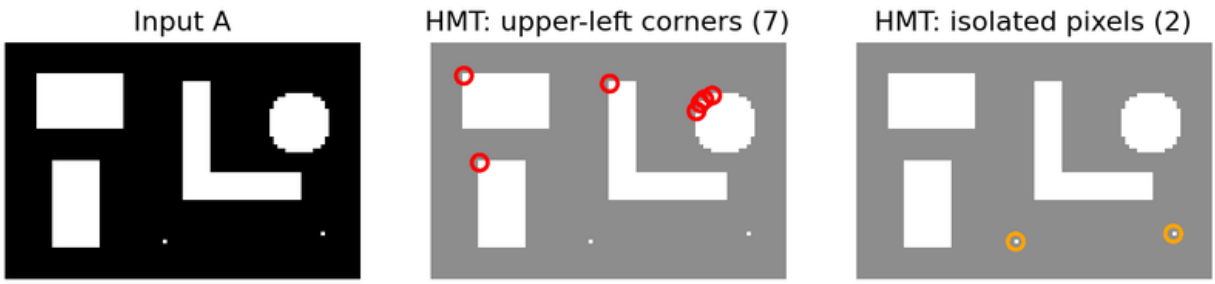


Figure 9.2 — The hit-or-miss transform as an exact pattern detector. Middle: upper-left corners (including small steps on the digital disk). Right: isolated foreground pixels.

```
import numpy as np
from scipy import ndimage as ndi

A = np.zeros((7, 9), bool)
A[1:5, 1:6] = True                # a rectangle
A[5, 7] = True                    # an isolated pixel
hit  = np.array([[0, 0, 0], [0, 1, 1], [0, 1, 0]], bool)   # must be foreground
miss = np.array([[0, 1, 0], [1, 0, 0], [0, 0, 0]], bool)   # must be background
corners = ndi.binary_hit_or_miss(A, structure1=hit, structure2=miss)
print(np.argwhere(corners))       # [[1 1]]: the upper-left corner only
```

The HMT is the building block of thinning below; it also drives thickening and a purely morphological convex hull construction (both in the Supplementary notes).

Some basic morphological algorithms

In plain words: with erosion, dilation and the HMT you can build many practical tools. Most of them are short loops that repeat one operation until nothing changes.

Boundary extraction

The inner boundary of a set is what erosion removes:

$$\beta(A) = A - (A \ominus B),$$

where B is a 3×3 square (giving an 8-connected boundary) or a cross (giving a thicker, 4-connected one). Using a larger B gives a thicker boundary. The first panel of Figure 9.3 shows $\beta(A)$ for the horse.

Hole filling

A **hole** is a background region completely surrounded by foreground. Given one seed pixel inside a hole, grow the seed by repeated dilation but clip it to the background at every step:

$$X_k = (X_{k-1} \oplus B) \cap A^c, \qquad k = 1, 2, 3, \ldots$$

Here X_0 contains only the seed, B is the 4-connected cross, and the intersection with A^c stops the growth at the hole's wall. Stop when $X_k = X_{k-1}$; then $X_k \cup A$ is the filled shape. This "dilate, then clip" pattern is called **conditional dilation**, and it reappears below as geodesic dilation. Using the cross (not the square) as B matters: with an 8-connected foreground wall, the 4-connected background fill cannot leak diagonally through the wall.

```
import numpy as np
from scipy import ndimage as ndi

def fill_hole(A, seed):
    """Grow X from a seed inside one hole: X_k = dilate(X_{k-1}) AND not A."""
    B = np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]], bool)   # 4-connected cross
    X = np.zeros_like(A); X[seed] = True
    while True:
        X_next = ndi.binary_dilation(X, structure=B) & ~A
        if np.array_equal(X_next, X):
            return A | X
        X = X_next

yy, xx = np.mgrid[-12:13, -12:13]
ring = (np.hypot(yy, xx) <= 10) & (np.hypot(yy, xx) > 6)   # boolean ring
print(ring.sum(), fill_hole(ring, (12, 12)).sum())        # 204 -> 317
```

Extraction of connected components

Swap the roles: to extract the connected component of A that contains a seed pixel, grow the seed by dilation but clip it to the *foreground*:

$$X_k = (X_{k-1} \oplus B) \cap A.$$

With B a 3×3 square the result is the seed's 8-connected component; with a cross it is the 4-connected component. Repeating this for every unlabelled seed labels all components. Real libraries use faster one- or two-pass labelling algorithms, but the result is the same: `scipy.ndimage.label(A, structure=np.ones((3, 3)))` returns a label image and the number of components.

Convex hull

In plain words: the convex hull is the shape you get by stretching a rubber band around an object. A set is **convex** if the straight segment between any two of its points stays inside it, and the **convex hull** $C(A)$ is the smallest convex set containing A. It fills every concavity of the shape (second panel of Figure 9.3). Morphology can build it from hit-or-miss templates, but in practice `skimage.morphology.convex_hull_image` computes the polygonal hull of the foreground pixel coordinates and rasterises it, which is faster and closer to the Euclidean hull.

Online only: Convex hull by hit-or-miss iteration — https://blog.shyandram.dev/en/learn/image-processing/morphology/notes/convex-hull-by-hit-or-miss-iteration/

Thinning

Thinning removes boundary pixels selectively, without breaking the shape apart, until only a one-pixel-wide line remains. One thinning step with template B is

$$A \otimes B = A - (A \circledast B),$$

which deletes exactly the pixels matched by the HMT. A full thinning cycles through a sequence of templates $\{B^1, B^2, \dots, B^n\}$, usually 8 rotations of an "edge pixel" pattern,

$$A \otimes \{B\} = ((\cdots((A \otimes B^1) \otimes B^2)\cdots) \otimes B^n),$$

and repeats the whole cycle until no pixel changes. The templates are designed so that deleting a matched pixel never disconnects the shape and never removes the endpoint of a line. Zhang and Suen's parallel thinning algorithm is a widely used, efficient version of this idea [6]; `skimage.morphology.skeletonize` uses it by default for 2-D images, and `skimage.morphology.thin` implements a related parallel thinning.

Online only: Thickening — https://blog.shyandram.dev/en/learn/image-processing/morphology/notes/thickening/

Skeletons

In plain words: the skeleton is the thin centre line of a shape, a stick-figure version of it. Formally, it is defined with maximal disks: a point is on the **skeleton** $S(A)$ if it is the centre of a disk that fits inside A and is not contained in any other fitting disk. Such disks touch the boundary in at least two places. In practice you compute it by thinning, which gives a connected, one-pixel-wide skeleton, or as the **medial axis** (computed from a distance transform), which also attaches the radius of the maximal disk to each skeleton point, as in the right panel of Figure 9.3.

Online only: The morphological skeleton formula — https://blog.shyandram.dev/en/learn/image-processing/morphology/notes/the-morphological-skeleton-formula/

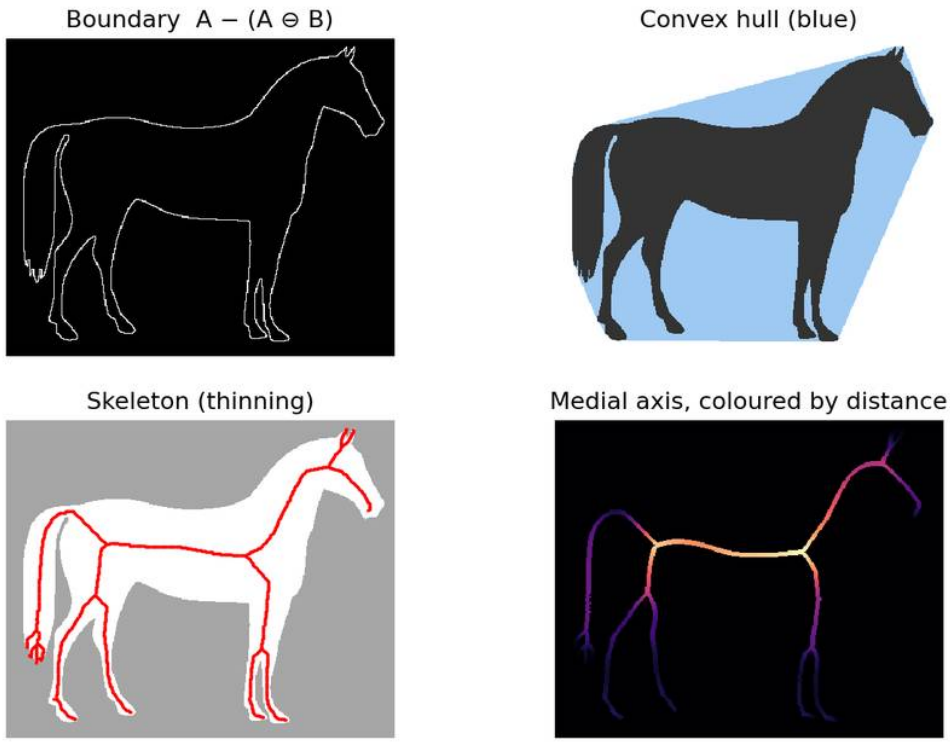


Figure 9.3 — Algorithms built from erosion and the HMT: inner boundary, convex hull, thinning skeleton, and medial axis whose colour encodes the local half-width.

Pruning

Skeletons are sensitive to boundary noise: a single bump on the outline sprouts a short **spur** (parasitic branch), visible at the horse's tail in Figure 9.3. **Pruning** removes spurs shorter than some length L:

1. Thin the skeleton by removing **end points** (pixels with exactly one 8-neighbour) L times. Every spur of length at most L disappears, but every genuine branch also loses L pixels at its tip.

2. Find the end points of the shortened skeleton.
3. Grow those end points back by L conditional dilations, constrained to the original skeleton, to restore the main branches to full length.
4. Take the union of the shortened skeleton and the regrown tips.

```
import numpy as np
from scipy import ndimage as ndi
from skimage import data, morphology

def endpoints(S):
    n = ndi.convolve(S.astype(int), np.ones((3, 3), int), mode="constant") - S
    return S & (n == 1)            # foreground pixels with exactly one neighbour

def prune_tips(S, L):
    """Step 1 of pruning: strip L layers of end points."""
    S = S.copy()
    for _ in range(L):
        S &= ~endpoints(S)
    return S

skel = morphology.skeletonize(~data.horse())
print(skel.sum(), prune_tips(skel, 10).sum())
```

Morphological reconstruction

In plain words: reconstruction lets you say "keep every object that contains this seed" and get the object back *exactly*, not a rounded copy of it. It uses two images: a **marker**, which says where to start, and a **mask**, which says how far you may grow.

Geodesic dilation and erosion

Let F be the marker and G the mask, with $F \subseteq G$. The **geodesic dilation of size 1** of F with respect to G is

$$D_G^{(1)}(F) = (F \oplus B) \cap G,$$

where B is a small SE (usually the 3×3 square). The marker grows by one step but may not leave the mask. Size n means n repetitions: $D_G^{(n)}(F) = D_G^{(1)}(D_G^{(n-1)}(F))$. The word *geodesic* refers to distances measured along paths that stay inside G, like walking inside a building rather than through its walls.

The dual **geodesic erosion of size 1** is

$$E_G^{(1)}(F) = (F \ominus B) \cup G,$$

with $F \supseteq G$: the marker shrinks, but never below the mask.

Reconstruction by dilation and by erosion

Iterate geodesic dilation until nothing changes. The result is the **reconstruction by dilation** of mask G from marker F:

$$R_G^D(F) = D_G^{(k)}(F) \quad \text{with } k \text{ such that } D_G^{(k)}(F) = D_G^{(k+1)}(F).$$

In binary images, $R_G^D(F)$ is exactly the union of the connected components of G that contain at least one marker pixel. Likewise, the **reconstruction by erosion** $R_G^E(F)$ iterates geodesic erosion to stability. Vincent's paper gives the standard efficient algorithms (queue-based, visiting each pixel a bounded number of times rather than iterating full-image passes) and many applications [7]; `skimage.morphology.reconstruction` follows that approach.

Opening by reconstruction

An ordinary opening removes small objects but also rounds the corners and erases the thin parts of the large ones. **Opening by reconstruction** avoids that:

$$O_R^{(n)}(F) = R_F^D(F \ominus nB),$$

where F is the input image, $F \ominus nB$ is n erosions by B (the marker), and the input itself is the mask. Erosion decides *which* objects survive (those that contain at least one placement of the SE); reconstruction restores each survivor to its original shape. Compare the "ordinary opening" and "opening by reconstruction" panels of Figure 9.4: the thin arm on the square and the sharp corners return intact. **Closing by reconstruction** is the dual, built from dilation and reconstruction by erosion.

Automatic hole filling

Reconstruction also fills all holes at once without seeds. Holes are background regions that cannot be reached from the image frame. So build a marker that is the complement of the image on the frame and zero elsewhere,

$$F(x,y) = \begin{cases} 1 - I(x,y) & \text{if } (x,y) \text{ is on the image border,} \\ 0 & \text{otherwise,} \end{cases}$$

reconstruct the background from it, and complement:

$$H = [R_{I^c}^D(F)]^c.$$

$R_{I^c}^D(F)$ is the background reachable from the border; everything else, H, is the foreground plus its holes.

Border clearing

Objects that touch the image border are often incomplete and should be excluded from measurements. Using the image itself on the border as the marker,

$$F(x,y) = \begin{cases} I(x,y) & \text{if } (x,y) \text{ is on the border,} \\ 0 & \text{otherwise,} \end{cases} \qquad X = I - R_I^D(F),$$

the reconstruction recovers every object that touches the border, and subtracting it leaves only interior objects.

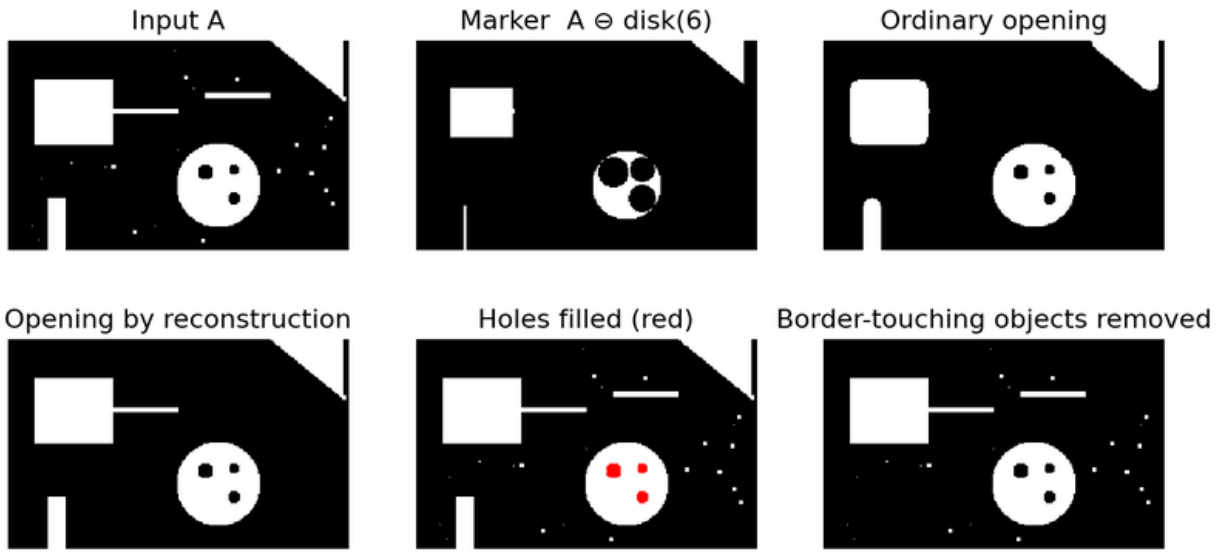


Figure 9.4 — Reconstruction. The eroded marker (disk of radius 6) picks the large objects; reconstruction restores them exactly, including the thin arm. Bottom: automatic hole filling and border clearing.

```
import numpy as np
from skimage import morphology, segmentation

def open_by_reconstruction(A, footprint):
    marker = morphology.binary_erosion(A, footprint)
    return morphology.reconstruction(marker.astype(np.uint8), A.astype(np.uint8),
                                     method="dilation").astype(bool)

def fill_holes_auto(A):
    """Background reachable from the image frame is background; the rest is hole."""
    Ac = ~A
    marker = np.zeros_like(Ac)
    marker[0, :], marker[-1, :] = Ac[0, :], Ac[-1, :]
    marker[:, 0], marker[:, -1] = Ac[:, 0], Ac[:, -1]
    reached = morphology.reconstruction(marker.astype(np.uint8), Ac.astype(np.uint8),
                                        method="dilation").astype(bool)
    return ~reached
```

```
# Library shortcuts: scipy.ndimage.binary_fill_holes(A), segmentation.clear_border(A)
```

Summary of binary operations

In plain words: every binary tool in this chapter is built from erosion and dilation plus set operations (complement, union, intersection). The ones to remember are opening and closing for cleaning, the hit-or-miss transform for exact patterns, and reconstruction for keeping whole objects exactly.

Online only: Table of binary operations — https://blog.shyandram.dev/en/learn/image-processing/morphology/notes/table-of-binary-operations/

Grayscale morphology

In plain words: for gray images, "fits inside" becomes "local minimum" and "touches" becomes "local maximum". Picture the image as a landscape where brightness is height. Erosion lowers each pixel to the lowest ground under the stencil; dilation raises it to the highest.

Grayscale erosion and dilation

With a **flat** SE b (a set of offsets, all of height zero), grayscale erosion and dilation of an image f are

$$[f \ominus b](x,y) = \min_{(s,t)\in b} f(x+s,\ y+t), \qquad [f \oplus b](x,y) = \max_{(s,t)\in b} f(x-s,\ y-t),$$

where (s,t) runs over the SE's offsets. The minus sign in dilation is the reflection again. On a binary image these reduce exactly to the set definitions. Erosion darkens the image, shrinks bright features and widens dark ones; dilation does the opposite.

Erosion and dilation can also use a **nonflat** SE that carries a height at each offset, but such SEs are rarely used in practice because the result depends on the image's intensity scale.

Online only: Nonflat structuring elements — https://blog.shyandram.dev/en/learn/image-processing/morphology/notes/nonflat-structuring-elements/

Opening and closing

The definitions are unchanged: $f \circ b = (f \ominus b) \oplus b$ and $f \bullet b = (f \oplus b) \ominus b$. The geometric picture is a ball pushed up from *beneath* the intensity surface: the opening is the highest surface the ball can reach while staying below f. Bright peaks narrower than the ball are cut down; everything wider is left alone. Closing pushes the ball down from *above*: dark valleys narrower than the ball are filled. The properties (anti-extensive/extensive, increasing, idempotent) carry over with $\leq$ in place of $\subseteq$.

Morphological smoothing

Because opening suppresses small bright details and closing suppresses small dark details, the sequence **opening followed by closing** with the same SE removes both kinds of small structure, while keeping sharp edges of larger regions. This is the grayscale version of the noise filter in Figure 9.1. A related scheme, **alternating sequential filtering**, applies opening–closing pairs with SEs of increasing size, which avoids the artefacts of using one large SE at once.

Morphological gradient

Dilation minus erosion gives a measure of local contrast:

$$g = (f \oplus b) - (f \ominus b).$$

Inside uniform regions the max and min are nearly equal, so $g \approx 0$; across an edge they differ by the edge height. Unlike a Sobel derivative, g is non-negative and direction-independent (for a symmetric b). Using only one side gives the **internal gradient** $f - (f \ominus b)$ or the **external gradient** $(f \oplus b) - f$, which place the edge response just inside or just outside the brighter region.

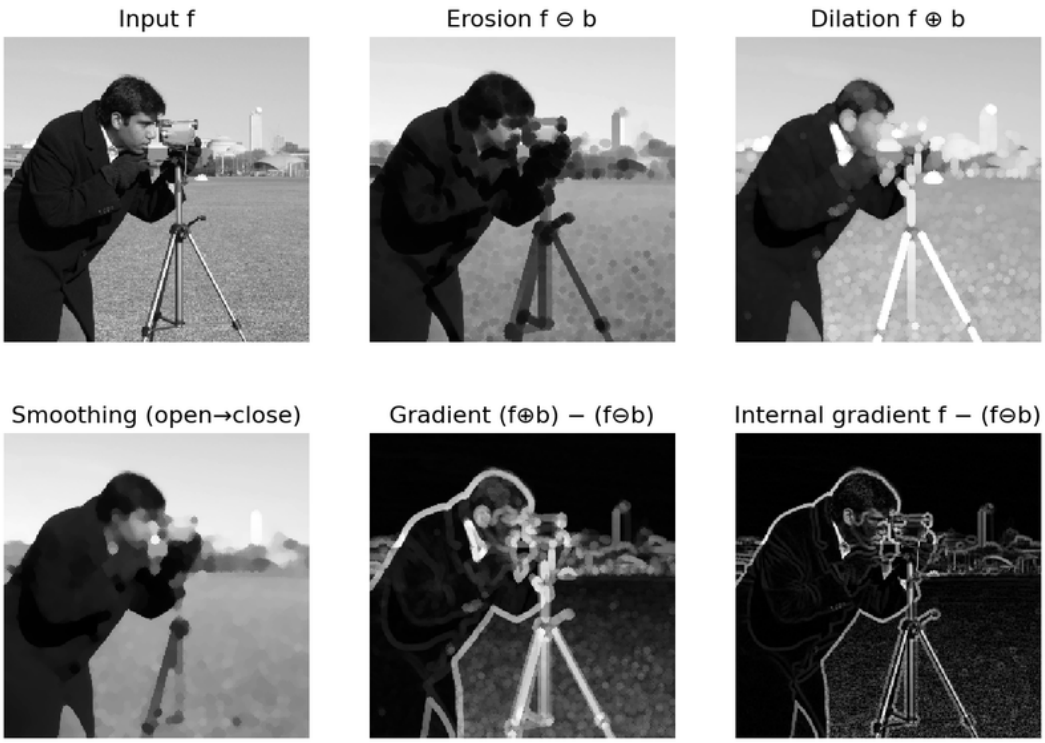


Figure 9.5 — Grayscale morphology with a flat disk of radius 3: erosion (local min), dilation (local max), smoothing by opening then closing, the morphological gradient, and the internal gradient.

```
from skimage import data, morphology, util
```

```
f = util.img_as_float(data.camera())
b = morphology.disk(3)                          # flat structuring element
ero  = morphology.erosion(f, b)                 # local minimum
dil  = morphology.dilation(f, b)                # local maximum
grad = dil - ero                                # morphological gradient (>= 0)
smooth = morphology.closing(morphology.opening(f, b), b)
```

Top-hat and bottom-hat transforms

Opening with an SE larger than the bright objects removes those objects but keeps the slowly varying background. Subtracting the opening from the image therefore leaves only the objects. This is the **top-hat** (or white top-hat) transform:

$$T_{\text{hat}}(f) = f - (f \circ b).$$

Its dual, the **bottom-hat** (or black top-hat) transform, extracts dark objects on a lighter background:

$$B_{\text{hat}}(f) = (f \bullet b) - f.$$

Both are non-negative. The main use is correcting **uneven illumination** before thresholding. In Figure 9.6, a synthetic image of grains lit from one side defeats a global Otsu threshold (IP 10): the bright side of the background is brighter than the grains on the dark side. After a top-hat with a disk of radius 8 (larger than any grain), the background is flat and a single threshold separates the grains almost perfectly. The SE size is the only parameter: it must be larger than the objects and smaller than the scale over which the background changes.

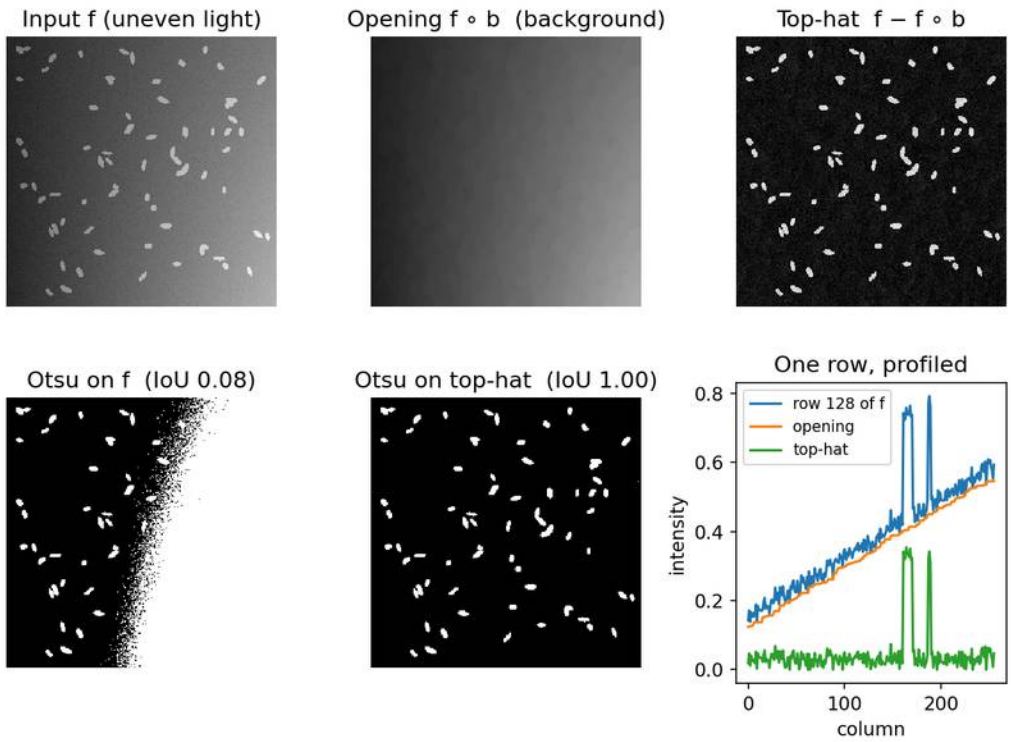
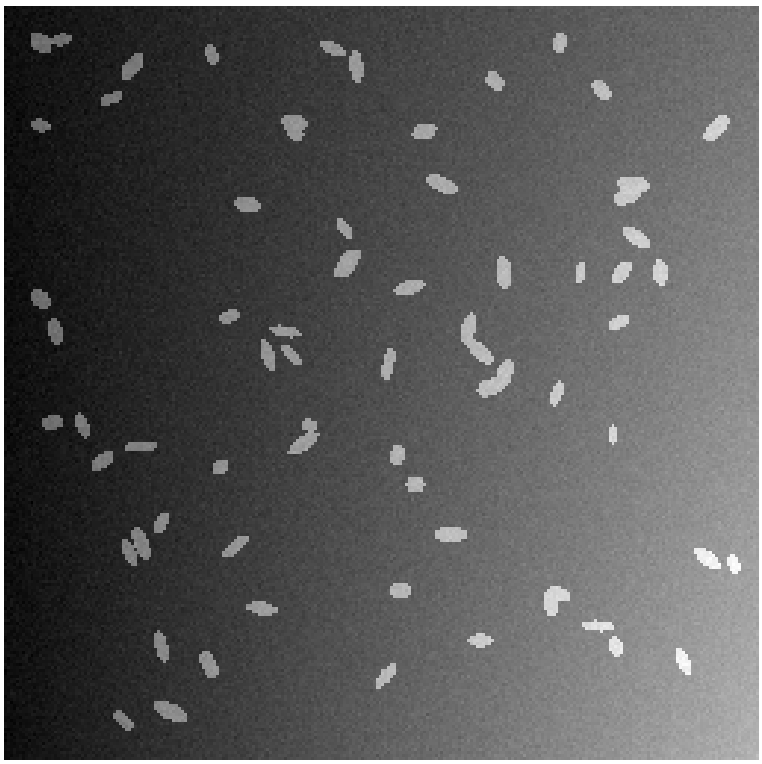


Figure 9.6 — Top-hat correction of uneven illumination. The opening with a disk larger than the grains is a background estimate; subtracting it flattens the background. IoU is measured against the known grain mask.

Drag the slider to compare the input with its top-hat:



Input



Top-hat (disk r=8)

Grains under uneven illumination, before and after the white top-hat transform

```
from skimage import data, filters, morphology, util

coins = util.img_as_float(data.coins())
th = morphology.white_tophat(coins, morphology.disk(30))   # bright objects
bh = morphology.black_tophat(coins, morphology.disk(30))   # dark objects
mask = th > filters.threshold_otsu(th)
```

Granulometry

Granulometry measures the size distribution of particles in an image without segmenting them. Open the image with SEs of increasing size r; each opening removes the bright particles smaller than b_r. Record the total intensity that remains,

$$V(r) = \sum_{x,y} [f \circ b_r](x,y), \qquad r = 0, 1, 2, \ldots,$$

where b_r is a disk of radius r and b_0 is a single pixel (so $V(0)$ is the sum of f). $V(r)$ never increases, because larger disks fit in fewer places. Its negative discrete derivative,

$$PS(r) = V(r-1) - V(r),$$

is a size histogram called the **pattern spectrum** [8]: a large value at r means a lot of image "mass" lives in features of radius about r. Figure 9.7 shows a scene with two particle sizes and the two corresponding peaks. The idea of a family of openings indexed by size, with the "sieve" interpretation, goes back to Matheron [2]; Maragos developed the pattern spectrum as a multiscale shape descriptor [8].

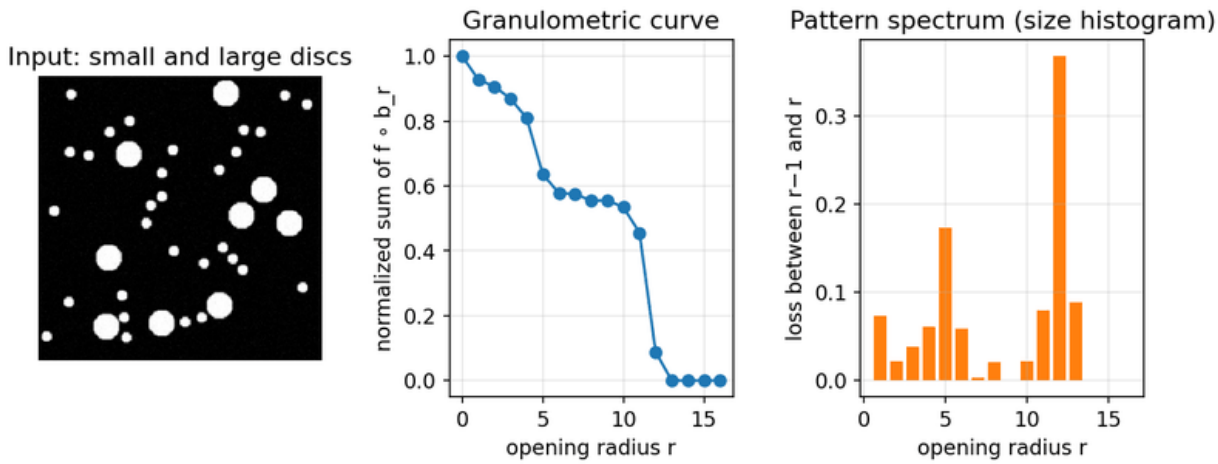


Figure 9.7 — Granulometry of a scene with discs of radius 5 and 12. The granulometric curve drops twice, and the pattern spectrum peaks at the two radii.

```
import numpy as np
from skimage import morphology

def granulometry(f, radii):
    total = f.sum()
    vol = np.array([morphology.opening(f, morphology.disk(r)).sum() / total
                    for r in radii])
    spectrum = -np.diff(vol)                # loss from radius r-1 to r
    return vol, spectrum
```

Online only: Textural segmentation — https://blog.shyandram.dev/en/learn/image-processing/morphology/notes/textural-segmentation/

Online only: Grayscale reconstruction — https://blog.shyandram.dev/en/learn/image-processing/morphology/notes/grayscale-reconstruction/

Modern view

Morphology is old, but it is far from finished. Five lines of work are worth knowing.

Foundations and tutorials. Matheron's work on random sets [2] and Serra's book [3] set up the algebra: erosion and dilation as Minkowski operations and openings as "sieves". Haralick, Sternberg and Zhuang's 1987 tutorial [4] is still one of the clearest introductions for engineers, and Soille's *Morphological Image Analysis* [5] is the practitioner's reference for geodesic transforms, reconstruction, watersheds, granulometries and efficient implementations.

Online only: Foundations of mathematical morphology — https://blog.shyandram.dev/en/learn/image-processing/morphology/papers/foundations-of-mathematical-morphology/

Reconstruction and efficient algorithms. Vincent's 1993 paper [7] made reconstruction practical. Its main takeaway is that the naive "iterate until stable" loop can be replaced by queue-based algorithms whose cost grows roughly linearly with the image size, which turned reconstruction into an everyday tool for hole filling, regional extrema, and h-domes.

Connected operators and component trees. A filter is **connected** if it can only remove or merge flat zones (connected regions of constant value) and never creates new contours; opening by reconstruction is the simplest example. Salembier and Wilkinson's review [10] explains the family, which includes the attribute openings of Breen and Jones [9]: they keep or remove a whole component by its area or shape rather than by whether an SE fits. Fast implementations filter a tree of the image's threshold components, the **max-tree** [11]; scikit-image exposes these as `area_opening`, `area_closing`, `diameter_opening` and `max_tree`.

Online only: Connected operators and component trees — https://blog.shyandram.dev/en/learn/image-processing/morphology/notes/connected-operators-and-component-trees/

Size distributions. Maragos's pattern spectrum [8] formalised granulometry as a multiscale shape descriptor; it is still used for texture and particle-size analysis, and attribute-based granulometries [9] extend it to shape attributes other than size.

Morphology meets deep learning. Deep learning has changed morphology in two directions.

Morphology inside networks. Max pooling is already a flat grayscale dilation followed by subsampling, so the connection is natural. Several groups have built network layers whose structuring elements are learned by gradient descent [13], [14]; because min and max have sparse gradients, later work replaces them with smooth, differentiable approximations [15], [16]. These remain research tools: morphological layers are not a standard component of mainstream vision backbones.

Online only: Learnable morphological layers — https://blog.shyandram.dev/en/learn/image-processing/morphology/notes/learnable-morphological-layers/

Morphology around networks. In segmentation, morphology is used constantly as **post-processing** of predicted masks: opening to remove false-positive specks, closing or hole filling to repair fragmented objects, `remove_small_objects`, border clearing, and skeletons for measuring tubular structures. It also appears inside **loss functions**. clDice [17] computes a differentiable "soft skeleton" by iterating min- and max-pooling (soft erosion and dilation), and uses overlap between the predicted and true skeletons as a loss that encourages topologically correct segmentation of vessels, roads and neurons.

What has *not* changed: for cleaning, measuring and correcting binary or grayscale images, the classical operators of this chapter are still the fastest, most predictable tools, and they come with guarantees (idempotence, connectivity, exact shape preservation) that learned filters do not offer.

Key takeaways

- Morphology works with **shape**, not frequency. It uses only min, max, union and intersection, controlled by a structuring element whose size and shape encode the question you ask.
- **Erosion** keeps positions where the SE fits; **dilation** keeps positions where the reflected SE touches. They are duals: eroding the foreground equals dilating the background.
- **Opening** (erode then dilate) removes small bright parts; **closing** (dilate then erode) fills small dark gaps. Both are idempotent filters, and opening followed by closing is a robust noise cleaner.
- The **hit-or-miss transform** matches an exact foreground/background pattern and underlies thinning, thickening, and convex hulls.
- **Reconstruction** (marker plus mask, grow until stable) keeps whole objects exactly: opening by reconstruction, automatic hole filling and border clearing all follow from it.
- In grayscale, erosion and dilation are local min and max. The **gradient**, **top-hat** and **bottom-hat** give edges, background correction, and small-object extraction; **granulometry** gives size distributions without segmentation.
- Today morphology lives on as post-processing for neural-network masks, as differentiable building blocks in losses such as clDice, and as a research topic in learnable morphological layers.

Exercises

1. A binary image contains a rectangle 40 pixels wide and 10 pixels tall. Describe its erosion and its opening by (a) a 5×5 square, (b) a horizontal line 15 pixels long, (c) a vertical line 15 pixels long. Which SE removes the rectangle entirely, and why?

   Hint

   Erosion keeps the positions of the SE's origin where the whole SE fits. The 5×5 square fits at a 36×6 block of positions, and the opening restores the full rectangle, because the rectangle is a union of fitting squares. The horizontal line also fits, so its opening is again the full rectangle. The vertical line is taller than the rectangle and never fits: both its erosion and its opening are empty.

2. Show that dilation is commutative and associative for sets: $A \oplus B = B \oplus A$ and $(A \oplus B) \oplus C = A \oplus (B \oplus C)$. Use the second identity to explain why dilating twice with a 3×3 square equals dilating once with a 5×5 square. Is the same true for erosion?

   Hint

   Write dilation as the Minkowski sum $\{a + b \mid a \in A, b \in B\}$; both identities follow from the commutativity and associativity of vector addition. A 3×3 square plus itself is a 5×5 square. For erosion, use $(A \ominus B) \ominus C = A \ominus (B \oplus C)$, which follows from duality.

3. Design the hit-or-miss template pair (B_1, B_2) that detects the **end points** of a one-pixel-wide 8-connected line pointing to the right (the line arrives from the left). How many rotated templates do you need to catch all end points? Test your answer with `scipy.ndimage.binary_hit_or_miss` on `skimage.morphology.skeletonize(~skimage.data.horse())`.

   Hint

   Centre in B_1 plus the left neighbour in B_1; the other seven neighbours in B_2. A line can also arrive diagonally, so you need 8 templates (4 axis directions and 4 diagonals). Compare your total count with the count from a neighbour-counting function like `endpoints()` in the pruning snippet.

4. Your segmentation network outputs masks of cells. Many masks have small holes, a few single-pixel false positives, and some cells are cut by the image border. Write a five-line post-processing pipeline in scikit-image and justify the order of the steps.

   Hint

   One reasonable order: `remove_small_objects` (or a small opening) first, so that specks do not get filled or merged; then `binary_fill_holes`; then a small closing if contours are ragged; then `clear_border`; finally `label` to count. Discuss what would go wrong if you closed before removing specks (nearby specks can merge into a "cell").

5. For a grayscale image f and a flat SE b, prove that the top-hat $T(f) = f - (f \circ b)$ is non-negative and that it is idempotent: $T(T(f)) = T(f)$.

Hint

Non-negativity follows from $f \circ b \le f$. For idempotence, show that $T(f) \circ b = 0$. Take any translate W of b and let y be the point of W where f is smallest. Since the opening is at least $\min_W f$ everywhere on W and at most f, it equals $f(y)$ at y, so $T(f)(y) = 0$. Every translate of b therefore contains a zero of $T(f)$, the erosion of $T(f)$ by b is zero, and so is its opening.

References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018, Ch. 9. publisher page
2. G. Matheron, *Random Sets and Integral Geometry*, Wiley, 1975. library record
3. J. Serra, *Image Analysis and Mathematical Morphology*, Academic Press, 1982. library record
4. R. M. Haralick, S. R. Sternberg, and X. Zhuang, "Image Analysis Using Mathematical Morphology," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. PAMI-9, pp. 532–550, 1987. doi
5. P. Soille, *Morphological Image Analysis*, Springer, 2004. doi
6. T. Y. Zhang and C. Y. Suen, "A Fast Parallel Algorithm for Thinning Digital Patterns," *Communications of the ACM*, vol. 27, pp. 236–239, 1984. doi
7. L. Vincent, "Morphological Grayscale Reconstruction in Image Analysis: Applications and Efficient Algorithms," *IEEE Trans. Image Processing*, vol. 2, pp. 176–201, 1993. doi
8. P. Maragos, "Pattern Spectrum and Multiscale Shape Representation," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 11, no. 7, pp. 701–716, 1989. doi
9. E. J. Breen and R. Jones, "Attribute Openings, Thinnings, and Granulometries," *Computer Vision and Image Understanding*, vol. 64, no. 3, pp. 377–389, 1996. doi
10. P. Salembier and M. H. F. Wilkinson, "Connected Operators: A Review of Region-Based Morphological Image Processing Techniques," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 136–157, 2009. doi
11. E. Carlinet and T. Géraud, "A Comparative Review of Component Tree Computation Algorithms," *IEEE Trans. Image Processing*, vol. 23, pp. 3885–3895, 2014. doi
12. S. van der Walt et al., "scikit-image: Image Processing in Python," *PeerJ*, 2014. arXiv
13. R. Mondal, S. Santra, S. S. Mukherjee, and B. Chanda, "Morphological Network: How Far Can We Go with Morphological Neurons?," BMVC, 2022. arXiv
14. K. Nogueira, J. Chanussot, M. Dalla Mura, and J. A. dos Santos, "An Introduction to Deep Morphological Networks," arXiv:1906.01751, 2019. arXiv
15. R. Hermary, G. Tochon, É. Puybareau, A. Kirszenberg, and J. Angulo, "Learning Grayscale Mathematical Morphology with Smooth Morphological Layers," *Journal of Mathematical Imaging and Vision*, vol. 64, pp. 736–753, 2022. doi
16. A. Kirszenberg, G. Tochon, É. Puybareau, and J. Angulo, "Going Beyond p-convolutions to Learn Grayscale Morphological Operators," arXiv:2102.10038, 2021. arXiv

17. S. Shit et al., “clDice — A Novel Topology-Preserving Loss Function for Tubular Structure Segmentation,” CVPR, 2021. arXiv

IP 10 • Image Segmentation I: Edge Detection, Thresholding, and Region Detection

Split an image into meaningful parts: find edges with gradients, LoG and Canny, link them with the Hough transform, pick thresholds with Otsu, and grow, cluster, cut and flood regions.

Online edition: https://blog.shyandram.dev/en/learn/image-processing/segmentation/

What you'll learn

- The formal definition of segmentation, and its two families: **discontinuity** (edges) and **similarity** (regions).
- How derivatives find points, lines and edges, and how Sobel, Marr–Hildreth (LoG) and Canny work step by step.
- How to link edge pixels into boundaries, including with the Hough transform.
- How Otsu's method picks a threshold, and what to do when one threshold is not enough.
- How region growing, split-and-merge, k-means, SLIC, normalized cuts, watersheds and motion turn pixels into regions.

The big picture

Until now, most operations took an image and returned an image. Segmentation returns a **map of parts**: each pixel gets a label saying which object or region it belongs to. Measuring objects, counting cells and reading text all depend on that map.

> **Explain like I'm five**
>
> Imagine a colouring page that has lost its black outlines. You can find the parts in two ways. Look for places where the colour suddenly changes and draw a line there: that is **edge detection**. Or start at one spot and keep spreading the same crayon while the paper looks the same: that is **region growing**. This chapter is about both tricks and how to make them work on messy real photos.

None of these methods learns from labelled examples. They need no training data, are easy to explain, and their ideas (gradients, hysteresis, between-class variance, graph cuts, flooding) live on inside modern deep pipelines.

Fundamentals

In plain words: segmentation cuts the image into pieces that do not overlap, cover everything, and each look "the same" inside.

Let R be the whole spatial region an image occupies. Segmentation partitions R into n subregions $R_1, \dots, R_n$ such that

(a) $\bigcup_{i=1}^{n} R_i = R$, (b) R_i is connected,
(c) $R_i \cap R_j = \emptyset$ for $i \neq j$, (d) $Q(R_i) = \text{TRUE}$, (e) $Q(R_i \cup R_j) = \text{FALSE}$ for adjacent R_i, R_j.

Here Q is a **logical predicate**: a yes/no test on a set of pixels, for example "all intensities lie within 10 grey levels of each other". So: every pixel is labelled (a), each region is one connected piece (b), no pixel has two labels (c), each region passes the test (d), and touching regions would fail it if merged (e).

Almost all monochrome segmentation algorithms rest on one of two properties of intensity [1]:

- **Discontinuity**: boundaries are where intensity changes abruptly. Point, line and edge detection use this.
- **Similarity**: pixels in one region share a property (intensity, colour, texture). Thresholding, region growing, clustering, graph cuts and watersheds use this.

Edges are precise but often broken; regions are closed by construction but their borders can wander.

Point, line, and edge detection

In plain words: an edge is where brightness changes fast, and "how fast something changes" is exactly what a derivative measures.

Background: derivatives on a pixel grid

For a 1-D digital function $f(x)$ the simplest approximations are

$$\frac{\partial f}{\partial x} \approx f(x+1) - f(x), \qquad \frac{\partial^2 f}{\partial x^2} \approx f(x+1) + f(x-1) - 2f(x).$$

Here x is an integer pixel position and $f(x)$ its intensity. Across a ramp, the first derivative is nonzero all along it, so it gives **thick** responses. The second derivative is nonzero only at the two ends, with opposite signs, giving a **double response** with a **zero crossing** in between, and it reacts far more strongly to fine detail and noise. These facts explain every detector below.

Detecting isolated points

A point is a tiny blob that differs from its surroundings. The Laplacian

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

is implemented with a 3×3 kernel whose centre is -8 and whose eight neighbours are 1 (the version that includes diagonals). With $Z(x, y)$ the filtered image, we mark a point wherever $|Z(x, y)| > T$ for a nonnegative threshold T. The kernel's coefficients sum to zero, so flat regions give no response.

Detecting lines

A line is a structure one or a few pixels wide. The Laplacian responds to lines too, but with a double response and no sense of direction. For a specific angle, use a kernel whose 2s run along that angle and whose other entries are -1 (the horizontal one has $2, 2, 2$ in its middle row). Comparing the responses of the horizontal, $+45°$, vertical and $-45°$ kernels tells which orientation dominates at each pixel.

Edge models

Real edges come in three idealized shapes:

- a **step edge** jumps from one level to another across one pixel (it appears mainly in synthetic images);
- a **ramp edge** changes linearly over several pixels, which is what blur from optics and sampling produces in real images;
- a **roof edge** rises and falls back, like a thin line seen through a blur.

On a ramp, the first-derivative magnitude says an edge is present, the sign of the second derivative says which side is bright, and its zero crossing marks the ramp's centre. But noise too faint to see ruins the second derivative, so **smoothing before differentiating** is part of every serious edge detector.

Basic edge detection with the gradient

The **gradient** of f at (x, y) is the vector

$$\nabla f = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \partial f/\partial x \\ \partial f/\partial y \end{bmatrix}, \qquad M(x, y) = \sqrt{g_x^2 + g_y^2}, \qquad \alpha(x, y) = \operatorname{atan2}(g_y, g_x).$$

g_x and g_y are the partial derivatives, M is the **gradient magnitude** (edge strength) and α the **gradient direction** (where intensity rises fastest); the edge runs perpendicular to α. M is often approximated by the cheaper $|g_x| + |g_y|$.

The derivatives are computed with small kernels. The classic operators are summarized in [1]. The **Roberts** cross operators use 2×2 diagonal differences. The **Prewitt** operators use 3×3 kernels with rows $[-1, 0, 1]$. The **Sobel** operators weight the centre row by 2, for example

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix},$$

a difference in one direction times a $[1, 2, 1]$ smoothing in the other. That built-in smoothing is why Sobel is usually preferred.

```
import cv2
import numpy as np
from skimage import data

f = data.camera().astype(np.float32) / 255.0
gx = cv2.Sobel(f, cv2.CV_32F, 1, 0, ksize=3)   # d/dx (columns)
gy = cv2.Sobel(f, cv2.CV_32F, 0, 1, ksize=3)   # d/dy (rows)
M = np.hypot(gx, gy)                           # gradient magnitude
alpha = np.arctan2(gy, gx)                     # gradient direction, radians
edges = M > 0.3 * M.max()                      # crude thresholded edge map
```

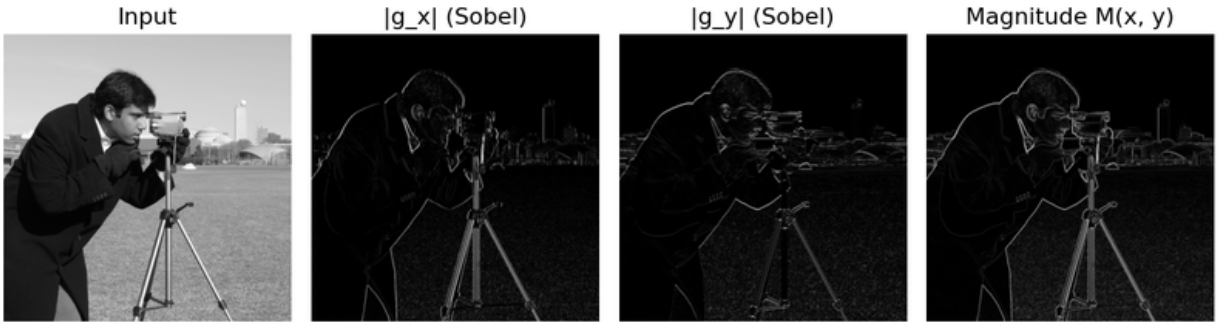


Figure 10.1 — Sobel derivatives. |g_x| lights up vertical structures such as the tripod legs, |g_y| lights up horizontal ones such as the horizon, and the magnitude M combines both. Note how texture in the grass also responds.

Try the Sobel kernel yourself. Swap it to the y version, or change the 2s to 1s to get Prewitt, and watch the response change.

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

Thresholding M gives a poor edge map: thick edges, lost weak edges, and clutter from texture. The next two detectors build smoothing and thinning into the design.

The Marr–Hildreth edge detector

Marr and Hildreth [2] argued that intensity changes occur at many scales, so a detector needs an adjustable size, and that second-derivative zero crossings mark edge centres. Their operator is the **Laplacian of a Gaussian (LoG)**, $\nabla^2 G$, where

$$G(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}}, \qquad \nabla^2 G(x,y) = \frac{x^2+y^2-2\sigma^2}{\sigma^4}\, e^{-\frac{x^2+y^2}{2\sigma^2}}.$$

σ is the Gaussian's standard deviation and sets the scale (larger σ, only coarser edges survive); the normalizing constant is dropped since only zero crossings matter. Its shape earns $\nabla^2 G$ the name **Mexican hat**. The algorithm:

1. Smooth with an $n \times n$ Gaussian, n the smallest odd integer $\geq 6\sigma$.
2. Take the Laplacian of the result (by linearity, steps 1–2 are one convolution with $\nabla^2 G$).
3. Mark **zero crossings**: pixels where some pair of opposite neighbours has different signs and an absolute difference above a threshold; without the threshold, every ripple in a flat area becomes an "edge".

A **difference of Gaussians (DoG)**, $G_{\sigma_1} - G_{\sigma_2}$ with $\sigma_1/\sigma_2 \approx 1.6$, approximates the LoG [1]. Zero crossings form closed contours, but they produce "spaghetti" in texture and round off corners.

The Canny edge detector

Canny [3] turned edge detection into an optimization problem with three criteria:

1. **Low error rate**: find all true edges and no spurious ones.
2. **Good localization**: the detected edge should be as close as possible to the true edge.
3. **Single response**: one true edge should produce one detected edge, not a double line.

For a 1-D step in white noise, the optimal filter is well approximated by the first derivative of a Gaussian. The practical 2-D algorithm follows:

1. **Smooth** the image with a Gaussian of standard deviation σ: $f_s = G_\sigma \star f$.
2. **Compute the gradient** magnitude M and angle α of f_s.
3. **Non-maximum suppression.** Quantize α into four directions. Zero any pixel whose M is smaller than either neighbour along the gradient direction. Ridges become one pixel thick.
4. **Double thresholding.** Pick T_H and T_L with T_H/T_L about 2 to 3 [1]. Pixels above T_H are **strong**; pixels between are **weak**.
5. **Hysteresis.** Keep a weak pixel only if it connects, through weak pixels, to a strong one.

Hysteresis is the key idea. One threshold either breaks contours or admits noise; two thresholds plus connectivity keep faint contours anchored to strong ones and drop isolated weak responses.

```
import cv2
from scipy import ndimage as ndi
from skimage import data, feature

f8 = data.camera()
canny_cv = cv2.Canny(cv2.GaussianBlur(f8, (0, 0), 2), 30, 90)      # uint8 0/255
canny_sk = feature.canny(f8 / 255.0, sigma=2, low_threshold=0.05, high_threshold=0.15)
log = ndi.gaussian_laplace(f8 / 255.0, sigma=3)                    # Marr-Hildreth steps 1-2
```

`cv2.Canny` has no smoothing parameter, so we blur first; scikit-image takes `sigma` directly.

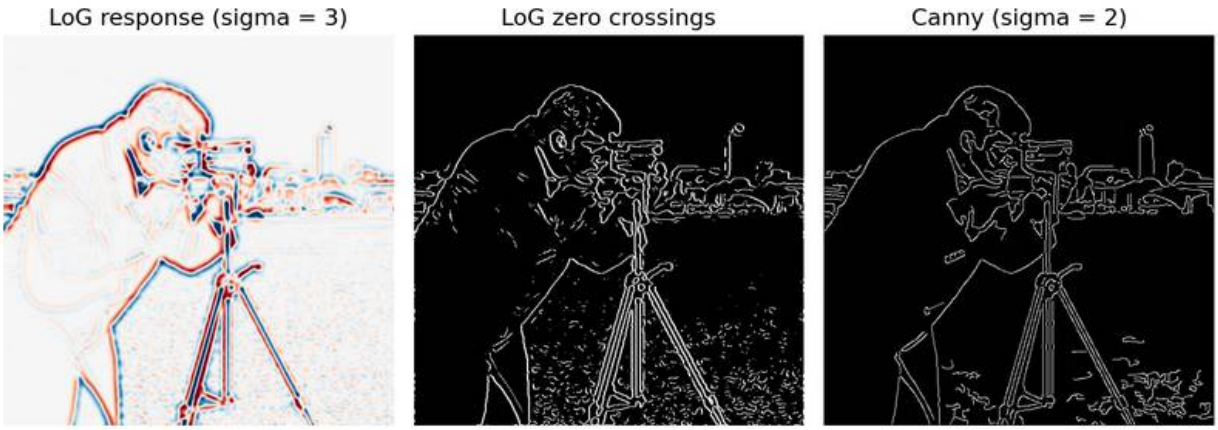


Figure 10.2 — Marr–Hildreth versus Canny on the same image. The LoG response (left, red positive, blue negative) changes sign at every edge. Its thresholded zero crossings (middle) are closed but cluttered in the grass. Canny (right) gives thin, mostly continuous edges with far less clutter.

Drag the slider to see which structures Canny keeps (edges are shown dark on white).



Input



Canny σ=2

Cameraman image compared with its Canny edge map

Linking edge points

Even Canny leaves gaps. **Edge linking** assembles edge pixels into meaningful boundaries.

Local processing. The simplest idea is to link neighbouring edge pixels whose gradient magnitude and direction are both similar, so that pixels on the same contour join up while unrelated ones stay apart.

Online only: Local edge linking — https://blog.shyandram.dev/en/learn/image-processing/segmentation/notes/local-edge-linking/

Global processing with the Hough transform. When the shape is known, let every edge pixel vote for all shapes that could pass through it. For lines, Duda and Hart [4] proposed the **normal representation**

$$\rho = x \cos\theta + y \sin\theta,$$

where θ is the angle of the line's normal and ρ its signed distance from the origin; unlike slope–intercept, this stays bounded for vertical lines. Each point (x_k, y_k) becomes a sinusoid in the $\rho\theta$ plane, and the sinusoids of collinear points meet in one cell. The algorithm:

1. Compute a binary edge map (e.g. Canny).

2. Quantize the $\rho\theta$ plane into **accumulator cells** $A(\rho, \theta)$, all zero.
3. For each edge pixel and each θ, compute ρ and increment $A(\rho, \theta)$.
4. Local maxima with large counts are lines; optionally check the continuity of their voters to get segments.

```
# continues from the Canny snippet above (uses cv2, np and canny_cv)
import numpy as np

lines = cv2.HoughLines(canny_cv, rho=1, theta=np.pi / 180, threshold=120)
for rho, theta in lines[:5, 0]:
    print(f"rho = {rho:6.1f} px, theta = {np.degrees(theta):5.1f} deg")
```

OpenCV reports $\theta \in [0°, 180°)$ with signed ρ, the same convention over a different range. Voting extends to circles and other shapes with more parameters.

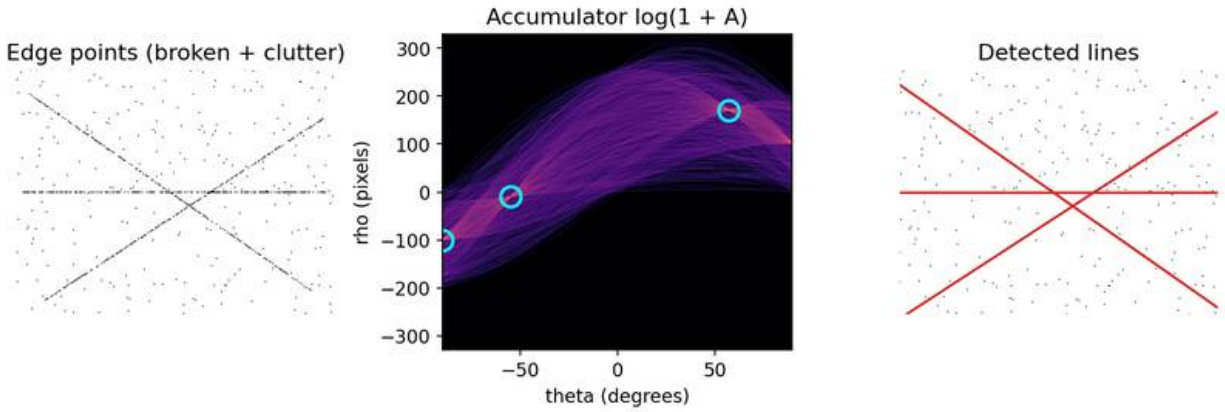


Figure 10.3 — The Hough transform. Three lines with 35% of their pixels missing, plus random clutter (left). Every point votes along a sinusoid; the three circled peaks in the accumulator (middle) are where many sinusoids meet. Converting the peaks back to lines recovers all three (right) despite the gaps.

Thresholding

In plain words: pick a grey level; everything brighter is "object" and everything darker is "background". The art is in picking the level.

Foundation

Given an intensity threshold T, global thresholding produces

$$g(x, y) = \begin{cases} 1 & \text{if } f(x, y) > T \\ 0 & \text{otherwise.} \end{cases}$$

A constant T is a **global** threshold; one that changes with position is **variable** (local, adaptive); several thresholds give **multiple thresholding**. It works when the histogram has well separated modes. **Noise** widens the modes, **uneven illumination and reflectance** shift them, and a tiny object makes a bump that is easy to miss.

Basic global thresholding

An easy iterative rule works well when the modes are clearly separated:

1. Pick an initial T, such as the mean intensity.
2. Split the pixels into G_1 (values $> T$) and G_2 (values $\leq T$).
3. Compute the mean of each group, m_1 and m_2.
4. Set $T = \frac{1}{2}(m_1 + m_2)$.
5. Repeat 2–4 until T changes by less than a small ΔT.

```
def basic_global(img, dT=0.5):
    T = img.mean()
    while True:
        m1, m2 = img[img > T].mean(), img[img <= T].mean()
        T_new = 0.5 * (m1 + m2)
        if abs(T_new - T) < dT:
            return T_new
        T = T_new
```

Otsu's optimum global thresholding

Otsu [5] picks the threshold that makes the two classes most **separable**, measured by the **between-class variance**. It needs only the histogram.

Let the image have L levels, n_i pixels at level i and MN pixels in total, so the normalized histogram is $p_i = n_i/MN$. A threshold k splits the levels into $C_1 = [0, k]$ and $C_2 = [k+1, L-1]$. Define

$$P_1(k) = \sum_{i=0}^{k} p_i, \qquad m(k) = \sum_{i=0}^{k} i\, p_i, \qquad m_G = \sum_{i=0}^{L-1} i\, p_i,$$

where $P_1(k)$ is the probability that a pixel falls in C_1, $m(k)$ is the cumulative mean up to level k, and m_G is the global mean. Then $P_2 = 1 - P_1$, and the class means are $m_1 = m/P_1$ and $m_2 = (m_G - m)/P_2$.

The **between-class variance** is the weighted spread of the two class means around the global mean. It simplifies to the product of the class probabilities times the squared gap between the class means:

$$\sigma_B^2 = P_1(m_1 - m_G)^2 + P_2(m_2 - m_G)^2 = P_1 P_2 (m_1 - m_2)^2.$$

The farther apart the class means, the larger σ_B^2. Substituting $m_1 = m/P_1$ and $m_2 = (m_G - m)/(1 - P_1)$ gives a form that needs only cumulative sums:

$$\sigma_B^2(k) = \frac{\left(m_G P_1(k) - m(k)\right)^2}{P_1(k)\left(1 - P_1(k)\right)}.$$

The optimum is $k^* = \arg\max_k \sigma_B^2(k)$ (average ties). Since the global variance σ_G^2 is the sum of between- and within-class variance and does not depend on k, this also **minimizes the within-class variance**. The ratio

$$\eta = \frac{\sigma_B^2(k^*)}{\sigma_G^2}, \qquad 0 \le \eta \le 1,$$

measures **separability**: near 1 for two clean modes, lower when they overlap.

Online only: Deriving the between-class variance — https://blog.shyandram.dev/en/learn/image-processing/segmentation/notes/deriving-the-between-class-variance/

```
import numpy as np
from skimage import data, filters

def otsu(img):
    p = np.bincount(img.ravel(), minlength=256) / img.size   # normalized histogram
    P1 = np.cumsum(p)                                         # class-1 probability
    m = np.cumsum(np.arange(256) * p)                         # cumulative mean
    mG = m[-1]                                                # global mean
    with np.errstate(divide="ignore", invalid="ignore"):
        sB2 = (mG * P1 - m) ** 2 / (P1 * (1 - P1))
    sB2 = np.nan_to_num(sB2)
    k = int(np.argmax(sB2))
    return k, sB2[k] / np.var(img)                            # threshold, separability

coins = data.coins()
k, eta = otsu(coins)          # k = 107, eta ≈ 0.76
assert k == filters.threshold_otsu(coins)
```

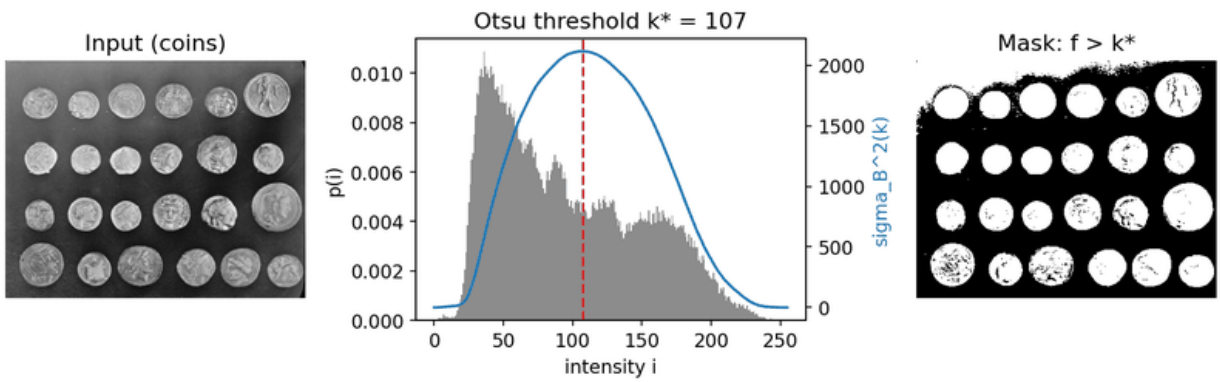


Figure 10.4 — Otsu's method on the coins image. The blue curve is σ_B²(k); its maximum gives k = 107 (dashed line). The mask (right) captures the coins, but the bright band along the top edge also passes, a reminder that one global threshold cannot correct uneven illumination.*

Using smoothing and edges to improve thresholding

- **Smooth first.** Noise widens the modes until they merge; smoothing narrows them again, as long as the object is large compared with the filter.
- **Use only pixels near edges.** A small object's mode is buried under the background's. Keep pixels where an edge indicator (gradient magnitude or $|\nabla^2 f|$) is high, say above its 99.7th percentile, and compute the histogram **from those pixels only**. They lie roughly half on each side of a boundary, so the histogram becomes balanced and bimodal. Apply the resulting threshold to the whole image.

Online only: Multiple thresholds — https://blog.shyandram.dev/en/learn/image-processing/segmentation/notes/multiple-thresholds/

Variable thresholding

When illumination varies, no single T works. Three remedies:

1. **Partitioning.** Threshold each tile with its own Otsu value; each tile must contain both classes.
2. **Local properties.** With m_{xy} and σ_{xy} the mean and standard deviation in a window around (x, y), use $T_{xy} = a\,\sigma_{xy} + b\,m_{xy}$ $(a, b \geq 0)$, or more generally a predicate $Q(\sigma_{xy}, m_{xy})$. Sauvola and Pietikäinen's document rule [7], $T_{xy} = m_{xy}(1 + k(\sigma_{xy}/R - 1))$ with R the dynamic range of σ and small $k > 0$, belongs to this family.
3. **Moving averages.** For text, scan in a zigzag and threshold each pixel at $c \cdot m(k)$, where $m(k)$ is the mean of the last n pixels and c is slightly below 1.

```
from skimage import data, filters

page = data.page()
global_mask = page > filters.threshold_otsu(page)
local_T = filters.threshold_local(page, block_size=35, offset=10)  # Gaussian-weighted local
↪  mean - 10
local_mask = page > local_T
sauvola_mask = page > filters.threshold_sauvola(page, window_size=25, k=0.2)
```

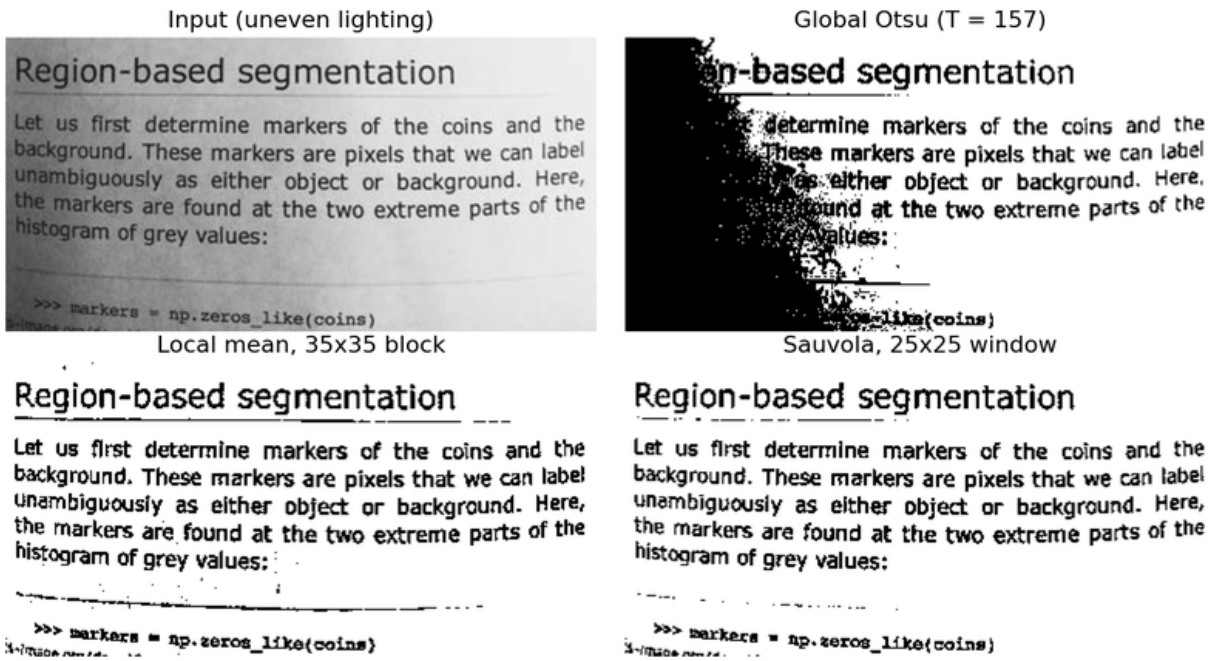


Figure 10.5 — Variable thresholding on unevenly lit text. A global Otsu threshold (top right) blacks out the shadowed left side. A local mean threshold (bottom left) and Sauvola's rule (bottom right) recover the text everywhere.

> **Interactive demonstration** (**online**): open the online edition of this tutorial to use this demonstration.Open demonstration

Region growing, splitting, and merging

In plain words: start from a few "seed" pixels and keep adding neighbours that look like them; or start from the whole image and keep cutting it into four until every piece is uniform.

Region growing

Region growing adds to each **seed** the neighbouring pixels that satisfy a similarity predicate. Choose the **seeds** (by hand, by a safe threshold, or from histogram peaks), the **predicate** (e.g. $|f(x, y) - f(\text{seed})| \le T$, or a test on region statistics), the **connectivity** (4 or 8), and the **stopping rule** (no neighbour passes; size or shape limits make it more robust).

```
from collections import deque
import numpy as np

def region_grow(img, seed, tol):
```

```
    """Grow from `seed` over 8-connected pixels whose value is within `tol` of the seed
     ↪  value."""
    H, W = img.shape
    ref = float(img[seed])
    mask = np.zeros((H, W), bool)
    mask[seed] = True
    q = deque([seed])
    while q:
        r, c = q.popleft()
        for dr in (-1, 0, 1):
            for dc in (-1, 0, 1):
                rr, cc = r + dr, c + dc
                if 0 <= rr < H and 0 <= cc < W and not mask[rr, cc] \
                        and abs(float(img[rr, cc]) - ref) <= tol:
                    mask[rr, cc] = True
                    q.append((rr, cc))
    return mask
```

`skimage.segmentation.flood(img, seed, tolerance=tol)` does the same in compiled code.

Online only: Region splitting and merging with quadtrees — https://blog.shyandram.dev/en/learn/image-processing/segmentation/notes/region-splitting-and-merging-with-quadtrees/

Region segmentation using clustering and superpixels

In plain words: treat every pixel as a point described by a few numbers (its colour, maybe its position) and gather nearby points into groups.

k-means clustering

Represent each pixel by a feature vector $\mathbf{z} \in \mathbb{R}^n$, for example its Lab colour ($n = 3$). **k-means** looks for k cluster centres $\mathbf{m}_1, \dots, \mathbf{m}_k$ and sets $C_1, \dots, C_k$ that minimize

$$J = \sum_{i=1}^{k} \sum_{\mathbf{z} \in C_i} \|\mathbf{z} - \mathbf{m}_i\|^2,$$

the total squared distance to the cluster means. The global minimum is NP-hard to find, so the standard algorithm alternates: assign each vector to its nearest centre, then move each centre to the mean of its vectors. It reaches a local minimum, so run it from several starts. Clustering on colour alone ignores position, so one cluster can be scattered over the image.

Superpixels and SLIC

A **superpixel** is a small, compact group of similar connected pixels. A few hundred superpixels instead of 10^5 pixels shrink later problems without moving important boundaries. **SLIC** by

Achanta et al. [8] is k-means in the 5-D space $[l, a, b, x, y]$. With N pixels and K superpixels, the grid spacing is $S = \sqrt{N/K}$.

1. Place K centres on a grid of spacing S, each nudged to the lowest gradient in its 3×3 neighbourhood.
2. Search **only a** $2S \times 2S$ **window** around each centre and assign pixels by

$$D = \sqrt{d_c^2 + \left(\frac{d_s}{S}\right)^2 m^2},$$

where d_c and d_s are Euclidean distances in Lab colour and in (x, y), and m is the **compactness**: large m gives square-ish superpixels, small m hugs colour edges. 3. Move each centre to the mean of its pixels; repeat about ten times. 4. **Enforce connectivity** by merging stray fragments into a neighbour.

The local search makes each iteration $O(N)$, independent of K.

```
from skimage import data, segmentation

img = data.astronaut()
sp = segmentation.slic(img, n_segments=300, compactness=10, start_label=1)
```

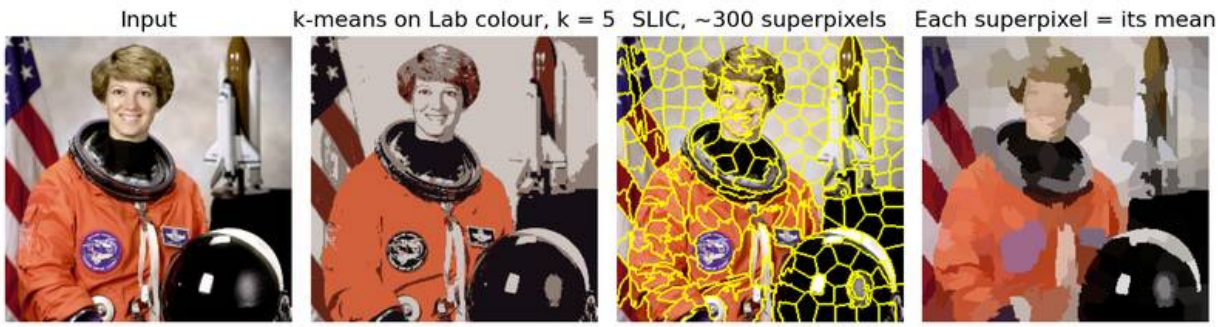


Figure 10.6 — Clustering. k-means on Lab colour (second panel) reduces the image to five colours, but each cluster is scattered across the picture. SLIC (third panel, compactness 10) produces about 300 compact superpixels that follow strong edges; painting each one with its mean colour (right) keeps the image readable at a tiny fraction of the pixel count.

Region segmentation using graph cuts

In plain words: draw the image as a net in which neighbouring pixels are tied together by strings, strong strings for similar pixels and weak strings for different ones. Segmenting is cutting the net into pieces while snapping only weak strings.

Images as graphs

A **graph** $G = (V, E)$ has nodes V and edges E. Each pixel (or superpixel) is a node; nearby nodes are joined by an edge with a nonnegative similarity weight $w(i, j)$, for example

$$w(i,j) = \exp\!\left(-\frac{\|\mathbf{F}_i - \mathbf{F}_j\|^2}{\sigma_I^2}\right)\exp\!\left(-\frac{\|\mathbf{X}_i - \mathbf{X}_j\|^2}{\sigma_X^2}\right) \quad \text{if } \|\mathbf{X}_i - \mathbf{X}_j\| \le r, \text{ else } 0,$$

where $\mathbf{F}$ is a feature vector (intensity, colour), $\mathbf{X}$ a spatial position, σ_I and σ_X scale parameters, and r a radius that keeps the graph sparse.

Minimum graph cuts

A **cut** splits V into two disjoint sets A and B by removing the edges between them. Its cost is

$$\operatorname{cut}(A, B) = \sum_{u \in A,\, v \in B} w(u, v).$$

The **minimum cut** removes as little similarity as possible. A popular formulation adds a **source** node (object) and a **sink** node (background), linked to every pixel with weights saying how much it resembles each. A minimum source–sink cut labels every pixel and can be found efficiently with maximum-flow algorithms; it underlies interactive tools where a user scribbles a few object and background pixels.

Normalized cuts

Without source and sink, a plain minimum cut prefers to cut off a single node or tiny group, because few edges are removed. Shi and Malik [9] fixed this with the **normalized cut**

$$\operatorname{Ncut}(A, B) = \frac{\operatorname{cut}(A, B)}{\operatorname{assoc}(A, V)} + \frac{\operatorname{cut}(A, B)}{\operatorname{assoc}(B, V)}, \qquad \operatorname{assoc}(A, V) = \sum_{u \in A,\, t \in V} w(u, t),$$

the total connection from A to all nodes. A tiny A has a tiny assoc, so cutting it off is now expensive. Exact minimization is NP-hard, but a relaxation gives an eigenproblem; with W the weight matrix and D the diagonal matrix of degrees $d_i = \sum_j w(i, j)$,

$$(D - W)\,\mathbf{y} = \lambda D\,\mathbf{y}.$$

The eigenvector of the **second smallest** eigenvalue is a soft partition indicator; thresholding it splits the graph, and recursion gives more segments. Practical code runs it on a superpixel graph.

```
import numpy as np
from skimage import data, segmentation, graph

img = data.coffee()
sp = segmentation.slic(img, n_segments=400, compactness=30, start_label=1)
```

```
rag = graph.rag_mean_color(img, sp, mode="similarity")   # edge weight = exp(-|c_i - c_j|^2 /
 ↪  sigma)
ncut = graph.cut_normalized(sp, rag, rng=0)              # recursive two-way Ncut
print(sp.max(), "superpixels ->", len(np.unique(ncut)), "regions")
```

Segmentation using morphological watersheds

In plain words: think of the image as a landscape where bright means high. Pour water in from every valley; where water from two valleys is about to meet, build a wall. The walls are the segmentation.

Background: basins and dams

Treat intensity as height. Each **regional minimum** has a **catchment basin**, the points whose water drains to it; **watershed lines** separate basins. We usually flood the **gradient magnitude**, not the image: interiors are low, boundaries are ridges, so each basin is one object.

Where two floods are about to meet, the algorithm builds a **dam**: a one-pixel-thick wall of pixels set higher than any intensity, so the floods never merge.

The watershed algorithm

In plain words: raise the water level one grey level at a time. At each level, a newly flooded patch either starts a new basin (a new minimum has appeared), joins the one basin it touches, or touches two or more basins, in which case a dam is built between them. Vincent and Soille [10] gave the efficient implementation used today: sort pixels by height, then flood level by level with a FIFO queue ("immersion simulation").

Online only: The flooding algorithm step by step — https://blog.shyandram.dev/en/learn/image-processing/segmentation/notes/the-flooding-algorithm-step-by-step/

The use of markers

On a raw gradient the watershed **over-segments**: noise and texture create thousands of minima. The cure is **markers**, components known in advance to be inside objects (**internal**) or background (**external**). Flooding starts only from markers, so there are as many regions as markers. Markers come from safe thresholds, smoothed minima, distance-transform peaks, or user clicks.

```
import numpy as np
from scipy import ndimage as ndi
from skimage import data, filters, morphology, segmentation

coins = data.coins()
```

```
elevation = filters.sobel(coins.astype(float))
markers = np.zeros_like(coins, dtype=np.int32)
markers[coins < 30] = 1        # sure background
markers[coins > 150] = 2       # sure object
ws = segmentation.watershed(elevation, markers)
coins_mask = morphology.remove_small_objects(ndi.binary_fill_holes(ws == 2), 100)
labels, n = ndi.label(coins_mask)   # n = 24 coins
```

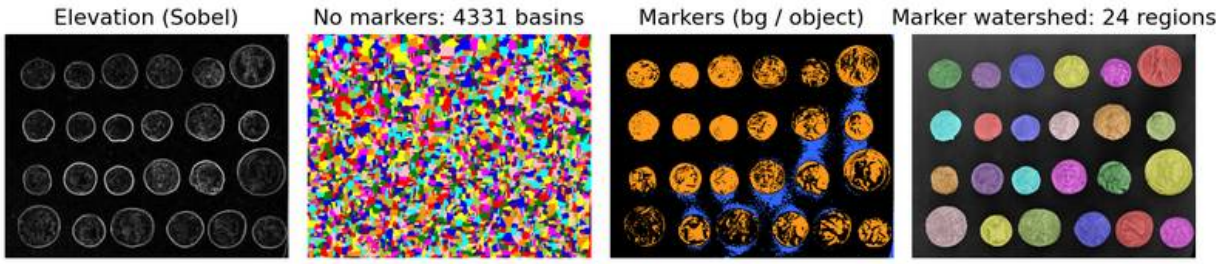


Figure 10.7 — Watershed on the coins image. Flooding the smoothed gradient from every regional minimum yields thousands of basins (second panel). With two marker labels from very safe thresholds (third panel), the floods meet exactly on the coin rims; filling holes and dropping specks leaves one region per coin (right).

The use of motion in segmentation

In plain words: if the camera is still, the things that moved between two frames are the things you care about. Subtract the frames and the moving objects light up.

Spatial techniques: difference images

Given two frames $f(x, y, t_i)$ and $f(x, y, t_j)$ of a static scene with a static camera, the **difference image** is

$$d_{ij}(x, y) = \begin{cases} 1 & \text{if } |f(x, y, t_i) - f(x, y, t_j)| > T \\ 0 & \text{otherwise,} \end{cases}$$

with T just above the noise level. Noise specks are removed by discarding small connected components.

Accumulative differences

One difference shows a moving object twice: where it was and where it is. An **accumulative difference image (ADI)** compares a **reference frame** $R = f(x, y, t_1)$ with each later frame and counts:

$$A_k = A_{k-1} + 1 \text{ if } |R - f_k| > T,$$
$$P_k = P_{k-1} + 1 \text{ if } R - f_k > T,$$
$$N_k = N_{k-1} + 1 \text{ if } R - f_k < -T,$$

each unchanged otherwise and starting at zero ($f_k = f(x, y, t_k)$). For an object brighter than the background, the **positive ADI** P marks where the object was in the reference frame; the **negative ADI** grows in the direction of motion, at a rate set by the speed; the **absolute ADI** A contains both.

```
import numpy as np

def accumulative_differences(frames, T):
    R = frames[0].astype(float)
    A = np.zeros(R.shape, int); P = np.zeros(R.shape, int); N = np.zeros(R.shape, int)
    for f in frames[1:]:
        d = R - f.astype(float)
        A += np.abs(d) > T
        P += d > T
        N += d < -T
    return A, P, N
```

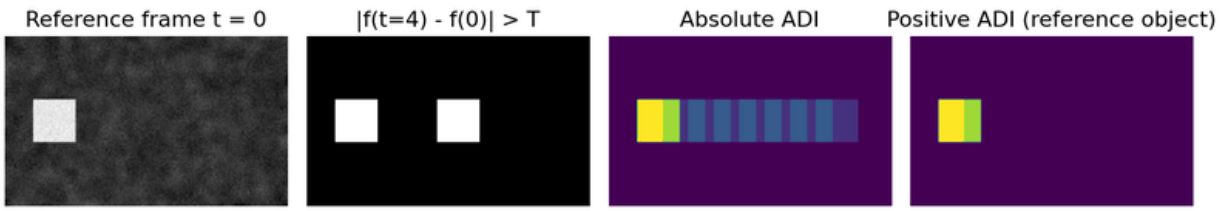


Figure 10.8 — Motion. A bright square moves right over eight frames. One difference image (second panel) shows the square twice. The absolute ADI (third panel) records every position it visited; the positive ADI (right) isolates where it started.

Online only: Building a reference image — https://blog.shyandram.dev/en/learn/image-processing/segmentation/notes/building-a-reference-image/

Modern view

Five surveys map the field; below each is what deep learning did and did not change.

Classical thresholding. Sezgin and Sankur [6] sort thresholding methods into six families by the information they exploit (histogram shape, measurement-space clustering, entropy, object attributes, spatial correlation, local grey-level surface) and evaluate forty of them on non-destructive testing and document images, singling out those that perform consistently in both. Otsu's method sits in the clustering family.

Deep segmentation. Minaee et al. [11] survey deep semantic and instance segmentation, grouping models into fully convolutional networks, encoder–decoders, multi-scale and pyramid

models, recurrent networks, attention models and adversarial generative models, and reviewing datasets and results. The landmark designs echo classical ideas:

- **FCN** [13] made classification networks fully convolutional so they output a label map, fusing coarse deep features with fine shallow ones: learned multi-scale analysis.
- **U-Net** [14] added a symmetric expanding path with skip connections, giving precise boundaries from little training data, and became a default in biomedical imaging. *Optional: U-Net: a reading guide to the paper* (*https://blog.shyandram.dev/en/learn/image-processing/segmentation/papers/u-net-a-reading-guide-to-the-paper/*) (*in the Paper guides*). *What the paper claims and how to read its tables.*
- **DeepLab** [16] used atrous convolution and atrous spatial pyramid pooling for resolution and context, plus a fully connected CRF to sharpen boundaries.
- **Mask R-CNN** [15] added a mask branch to an object detector, separating touching objects of one class, the job markers did for the watershed.

Foundation models and promptable segmentation. Segment Anything (SAM) [17] is a model prompted with points, boxes or masks, trained on SA-1B, over one billion masks on 11 million images. Click-in, mask-out is the interface of seeded region growing and interactive graph cuts, with learned instead of hand-made similarity. Zhou et al. [12] review over 300 methods of this "foundation model era", split into generic tasks (semantic, instance, panoptic) and promptable ones (interactive, referring, few-shot), and show how large pretrained models carry segmentation knowledge.

Edge detection. Sun et al. [18] group traditional edge detectors into gradient, Gaussian-difference, multi-scale and structured-learning methods, and deep ones into encoder–decoder, network-reconstruction and multi-scale fusion designs. HED [19], the key early deep design, fuses side outputs from several network depths into one edge map, a learned heir to Marr and Hildreth's multi-scale argument. The survey concludes that learned detectors now perform close to, or beyond, human level on standard benchmarks, leaving lightweight models, weak supervision and interpretability as open problems.

Superpixels. Stutz, Hermans and Leibe [20] benchmark 28 algorithms, stressing tuned parameters and **strictly enforced connectivity**, with metrics independent of the number of superpixels and robustness tests against noise, blur and affine transforms.

What changed and what did not. Learned features replaced hand-designed predicates and edge strengths, dramatically so on natural images. The classical skeleton remains: network outputs are probability maps that get thresholded; touching instances still need separating, by boxes, learned markers or a watershed on a predicted distance map; and Otsu, Canny and Hough still win where data are scarce, compute is tight or behaviour must be predictable, as in document scanning and industrial inspection.

Key takeaways

- Segmentation partitions an image into connected, non-overlapping regions that each satisfy a predicate Q; methods look either for **discontinuities** (edges) or for **similarity** (regions).

- First derivatives give thick edge responses; second derivatives give double responses with a zero crossing at the edge centre and are very sensitive to noise, so smooth before differentiating.
- Canny = Gaussian smoothing + gradient + non-maximum suppression + double threshold with **hysteresis**; the last step is what keeps faint but connected edges.
- The Hough transform turns line finding into voting in (ρ, θ) space, which is robust to gaps and clutter.
- Otsu's threshold maximizes the between-class variance $\sigma_B^2 = P_1 P_2 (m_1 - m_2)^2$, computable from cumulative histogram sums; use local thresholds when lighting is uneven.
- Region methods (growing, split-and-merge, k-means, SLIC, normalized cuts) group by similarity; normalized cuts avoid the small-segment bias of plain minimum cuts.
- The watershed floods a gradient image from its minima and builds dams where floods meet; **markers** are essential to avoid over-segmentation.
- Deep networks learned the similarity and edge measures, but thresholding, multi-scale fusion, seeds or prompts, and instance separation are still the skeleton of modern pipelines.

Exercises

1. A 1-D signal is flat at 10 for five samples, rises linearly to 50 over four samples, then stays at 50. Write down its first and second differences, and mark where the zero crossing of the second difference lies relative to the ramp.

   Hint

   The first difference is 10 on the four ramp steps, 0 elsewhere. The second difference is +10 at the ramp's start, −10 at its end, 0 between, so the zero crossing lies mid-ramp.

2. Show that maximizing the between-class variance σ_B^2 is equivalent to minimizing the weighted within-class variance $\sigma_W^2 = P_1 \sigma_1^2 + P_2 \sigma_2^2$.

   Hint

   Split $\sum_i (i - m_G)^2 p_i$ at k and add and subtract each class mean; cross terms vanish, leaving $\sigma_G^2 = \sigma_W^2 + \sigma_B^2$, with σ_G^2 independent of k.

3. In Canny's detector, what happens if you set $T_L = T_H$? What if $T_L = 0$? Predict the result, then test with `skimage.feature.canny` on `data.camera()`.

   Hint

   $T_L = T_H$ disables hysteresis: broken contours. $T_L = 0$ keeps every thinned pixel connected to a strong edge, so edges leak into the grass texture.

4. A Hough accumulator uses θ in 1° steps over $[-90°, 90°)$ and ρ in 1-pixel steps for a 640×480 image. How many accumulator cells are there, and how many increments does an edge map with 20,000 edge pixels cause?

Hint

$|\rho| \le \sqrt{640^2 + 480^2} = 800$, so about 1,601 ρ bins and 180 θ bins, which is about 288,000 cells. Each edge pixel votes once per θ: 20,000 × 180 = 3.6 million increments.

5. Explain why plain minimum cut tends to isolate single pixels, using a 4-connected pixel graph with uniform weights w. Then compute Ncut for isolating one interior pixel in an N-pixel image and show that it is close to 1, whereas a balanced cut along a short boundary scores close to 0.

Hint

Isolating one pixel costs $4w$, less than any long boundary. For Ncut, $\text{assoc}(A, V) = 4w$, so the first term is 1 and the second is tiny: Ncut ≈ 1, a bad score.

6. Run the marker-controlled watershed on `data.coins()` with background markers at `coins < 30` and object markers at `coins > 150`, as above, then change the object threshold to 120, 190 and 230. Which coins merge or vanish, and why?

Hint

At 120, object markers land on the bright top band, so coins merge with the background into big regions (we counted 11). At 190 there are still 24, since one marker pixel per coin suffices; at 230 darker coins lose their markers and vanish (13 left). Markers decide the result.

References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018, Ch. 10. publisher page
2. D. Marr and E. Hildreth, "Theory of edge detection," *Proceedings of the Royal Society of London. Series B*, vol. 207, no. 1167, pp. 187–217, 1980. doi
3. J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986. doi
4. R. O. Duda and P. E. Hart, "Use of the Hough transformation to detect lines and curves in pictures," *Communications of the ACM*, vol. 15, pp. 11–15, 1972. doi
5. N. Otsu, "A Threshold Selection Method from Gray-Level Histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, no. 1, pp. 62–66, 1979. doi
6. M. Sezgin and B. Sankur, "Survey over image thresholding techniques and quantitative performance evaluation," *Journal of Electronic Imaging*, vol. 13, no. 1, pp. 146–165, 2004. doi
7. J. Sauvola and M. Pietikäinen, "Adaptive document image binarization," *Pattern Recognition*, vol. 33, no. 2, pp. 225–236, 2000. doi
8. R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua and S. Süsstrunk, "SLIC Superpixels Compared to State-of-the-Art Superpixel Methods," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 11, pp. 2274–2282, 2012. doi

9. J. Shi and J. Malik, “Normalized cuts and image segmentation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 888–905, 2000. doi
10. L. Vincent and P. Soille, “Watersheds in digital spaces: an efficient algorithm based on immersion simulations,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, no. 6, pp. 583–598, 1991. doi
11. S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz and D. Terzopoulos, “Image Segmentation Using Deep Learning: A Survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 7, pp. 3523–3542, 2022. doi ·arXiv
12. T. Zhou, W. Xia, F. Zhang, B. Chang, W. Wang, Y. Yuan, E. Konukoglu and D. Cremers, “Image Segmentation in Foundation Model Era: A Survey,” arXiv:2408.12957, 2024. arXiv
13. J. Long, E. Shelhamer and T. Darrell, “Fully Convolutional Networks for Semantic Segmentation,” arXiv:1411.4038, 2014. arXiv
14. O. Ronneberger, P. Fischer and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation,” in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, Lecture Notes in Computer Science, vol. 9351, pp. 234–241, Springer, 2015. doi ·arXiv
15. K. He, G. Gkioxari, P. Dollár and R. Girshick, “Mask R-CNN,” arXiv:1703.06870, 2017. arXiv
16. L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy and A. L. Yuille, “DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 834–848, 2018. doi ·arXiv
17. A. Kirillov, E. Mintun, N. Ravi et al., “Segment Anything,” arXiv:2304.02643, 2023. arXiv
18. R. Sun, T. Lei, Q. Chen, Z. Wang, X. Du, W. Zhao and A. K. Nandi, “Survey of Image Edge Detection,” *Frontiers in Signal Processing*, 2022. doi
19. S. Xie and Z. Tu, “Holistically-Nested Edge Detection,” arXiv:1504.06375, 2015. arXiv
20. D. Stutz, A. Hermans and B. Leibe, “Superpixels: An Evaluation of the State-of-the-Art,” *Computer Vision and Image Understanding*, vol. 166, pp. 1–27, 2018. doi
21. I. Arganda-Carreras, S. C. Turaga, D. R. Berger et al., “Crowdsourcing the creation of image segmentation algorithms for connectomics,” *Frontiers in Neuroanatomy*, vol. 9, 2015. doi
22. M. Maška, V. Ulman, D. Svoboda et al., “A benchmark for comparison of cell tracking algorithms,” *Bioinformatics*, vol. 30, no. 11, pp. 1609–1617, 2014. doi ·Cell Tracking Challenge

IP 11 • Image Segmentation II: Active Contours —Snakes and Level Sets

Segment objects with curves that move: parametric snakes, gradient vector flow, balloon forces, and level-set methods such as geodesic active contours and Chan–Vese.

Online edition: https://blog.shyandram.dev/en/learn/image-processing/active-contours/

What you'll learn

- Why segmentation can be posed as moving a curve until an energy stops decreasing, and what that buys over thresholding and edge linking.
- How a **snake** (an explicit, parametric active contour) is defined by internal and external energies, and how the Euler–Lagrange equation turns that energy into an iterative algorithm.
- How to pick the external force: image gradients, edge maps, **gradient vector flow** (**GVF**) and the **balloon** force, and why each one exists.
- How the **level-set method** represents a curve implicitly as the zero crossing of a function ϕ, and how to evolve it with $\phi_t + F|\nabla\phi| = 0$ using upwind finite differences under a CFL limit.
- How the speed function encodes edge-based (**geodesic active contours**) and region-based (**Chan–Vese**, from Mumford–Shah) models, plus signed-distance reinitialization, narrow bands and morphological approximations.
- Where active contours live today: inside deep networks as contour-refinement heads and as loss functions.

The big picture

IP 10 segmented images with local rules: threshold this pixel, link that edge, grow this region. Those rules know nothing about the *shape* of what they produce. The result can be a boundary with gaps, a region that leaks through a weak edge, or a speckle of isolated pixels.

Active contours take a different view. We start with a closed curve somewhere near the object and let it **move**. The curve is pushed by the image (toward edges, or toward a good split into "inside" and "outside") and held together by its own stiffness (it resists stretching and bending). When the pushes and the stiffness balance, the curve stops, and its final position is the segmentation. The output is always a closed, smooth boundary, never a heap of edge fragments.

> **Explain like I'm five**
>
> Put a big rubber band around a toy on the table. Now let the band shrink. It slides inward until it hits the toy, and then it hugs the toy's outline. The rubber wants to be short and smooth; the toy stops it from shrinking any further. A snake is that rubber band, and the "toy" is wherever the picture has a strong edge. A level set is a cleverer band that can also snap into two bands if it finds two toys.

Following the textbook [1], this chapter covers the two families in that order. **Snakes** (Kass, Witkin and Terzopoulos [2]) store the curve as a list of points and move the points directly. **Level sets** (Osher and Sethian [5]) store the curve implicitly as the zero contour of a surface and move the surface instead, which makes splitting and merging automatic. Both are *variational*: we write down an energy, then do gradient descent on it. That idea, more than any particular formula, is what survived into deep learning.

Background: why curves that move

In plain words: instead of deciding pixel by pixel, we describe the whole boundary at once and ask, "how good is this boundary?" Then we improve it a little at a time.

More precisely, an active contour method has three ingredients:

1. A **representation** of a closed curve C in the image domain $\Omega \subset \mathbb{R}^2$: either explicit (a parametrized curve $v(s)$) or implicit (the zero level set of a function $\phi(x, y)$).
2. An **energy** $E(C)$ that is low when C is a good boundary. It always mixes a *regularity* term (short, smooth curves are preferred) with a *data* term (edges are strong along the curve, or the regions inside and outside are each homogeneous).
3. An **evolution rule** that changes C to decrease E, usually gradient descent written as a partial differential equation (PDE) in an artificial time t.

Compared with IP 10, the regularity term bridges gaps in edges, the output is always a closed curve ready for the shape descriptors of IP 12, and prior knowledge (a user's click, a smoothness preference, an intensity model) enters naturally.

The price is that the energy is non-convex: gradient descent finds a *local* minimum, so the answer depends on where you start.

Image segmentation using snakes

A snake is a closed chain of points that slides over the image. Each point feels two kinds of pull: its neighbours keep the chain short and smooth, and the image drags it toward edges.

The snake as a parametric curve

We write the contour as

$$v(s) = (x(s), y(s)), \qquad s \in [0, 1], \quad v(0) = v(1),$$

where s is a parameter that runs once around the curve and $x(s)$, $y(s)$ are image coordinates. Derivatives with respect to s are written $v_s = dv/ds$ and $v_{ss} = d^2v/ds^2$. In a computer the curve is sampled at n points $v_i = (x_i, y_i)$, $i = 0, \dots, n-1$, with indices taken modulo n for a closed snake.

The energy functional

In plain words: the energy gives every curve a score. It adds up how stretched and how bent the curve is, plus an image term that is low where we want the curve to settle; the snake looks for the curve with the lowest score.

The snake energy of [2] is

$$E_{\text{snake}}(v) = \int_0^1 \Big[\underbrace{\tfrac{1}{2}\big(\alpha\,|v_s(s)|^2 + \beta\,|v_{ss}(s)|^2\big)}_{E_{\text{int}}} + E_{\text{ext}}(v(s)) \Big]\, ds.$$

- $\alpha \geq 0$ is the **elasticity** (tension) weight. The term $|v_s|^2$ is large when the curve is stretched, so α makes the snake behave like an elastic string that wants to shrink.
- $\beta \geq 0$ is the **rigidity** (bending) weight. The term $|v_{ss}|^2$ measures how sharply the curve turns, so β makes the snake behave like a thin metal rod that resists kinks.
- $E_{\text{ext}}(x, y)$ is the **external (image) energy**, a scalar field over the image that is *low* where we want the snake to settle.

A common choice of external energy for a gray-level image f is

$$E_{\text{ext}}(x, y) = -\,w_{\text{edge}}\,|\nabla(G_\sigma * f)(x, y)|^2 \;-\; w_{\text{line}}\,(G_\sigma * f)(x, y),$$

where G_σ is a Gaussian of standard deviation σ, $*$ is convolution, ∇ is the gradient, and w_{edge}, w_{line} are weights. With $w_{\text{edge}} > 0$ the snake is attracted to strong edges. With $w_{\text{line}} > 0$ it is attracted to dark lines, and with $w_{\text{line}} < 0$ to bright lines. The Gaussian blur widens the valleys of E_{ext} so that the snake can feel an edge from a few pixels away; [2] calls this scale-space continuation.

From energy to motion: the Euler–Lagrange equation

To minimize this energy we use the calculus of variations: a minimizing curve must satisfy the **Euler–Lagrange equation** of the integrand. For constant α and β it reads

$$\alpha\, v_{ss} - \beta\, v_{ssss} - \nabla E_{\text{ext}}(v) = 0.$$

Read it as a force balance. αv_{ss} is the tension force (it points toward the local centre of curvature and shortens the curve), $-\beta v_{ssss}$ is the bending force (it straightens kinks), and $F_{\text{ext}} = -\nabla E_{\text{ext}}$ is the image force. At equilibrium they cancel.

We rarely can solve this equation directly, so we make v depend on an artificial time t and let it slide downhill:

$$\gamma\,\frac{\partial v}{\partial t} = \alpha\, v_{ss} - \beta\, v_{ssss} + F_{\text{ext}}(v),$$

where $\gamma > 0$ is a viscosity (damping) constant. When $\partial v/\partial t = 0$ we are back at the Euler–Lagrange equation.

Online only: Deriving the snake equation — https://blog.shyandram.dev/en/learn/image-processing/active-contours/notes/deriving-the-snake-equation/

Discretization and the iterative solution

In plain words: once the curve is sampled at points, the smoothness forces become one fixed matrix, so each step of the snake costs a single matrix–vector product per coordinate.

Sample the curve at n points with unit spacing. Central differences give

$$v_{ss} \approx v_{i+1} - 2v_i + v_{i-1}, \qquad v_{ssss} \approx v_{i+2} - 4v_{i+1} + 6v_i - 4v_{i-1} + v_{i-2}.$$

Stack the x coordinates into a vector $\mathbf{x} \in \mathbb{R}^n$ (and likewise $\mathbf{y}$). Then $\alpha v_{ss} - \beta v_{ssss} = -A\mathbf{x}$, where A is a symmetric, circulant, pentadiagonal $n \times n$ matrix whose rows contain

$$[\ \beta,\ \ -\alpha - 4\beta,\ \ 2\alpha + 6\beta,\ \ -\alpha - 4\beta,\ \ \beta\ ]$$

centred on the diagonal (wrapping around at the ends). The internal forces are linear in the point positions; the external force is not. The classic trick of [2] is a **semi-implicit** step: treat the internal part implicitly (at the new time t) and the external part explicitly (at the old time $t-1$):

$$\gamma\,(\mathbf{x}^t - \mathbf{x}^{t-1}) = -A\,\mathbf{x}^t + \mathbf{f}_x(\mathbf{x}^{t-1}, \mathbf{y}^{t-1}) \quad \Longrightarrow \quad \mathbf{x}^t = (A + \gamma I)^{-1}(\gamma\,\mathbf{x}^{t-1} + \mathbf{f}_x),$$

and the same for $\mathbf{y}$ with $\mathbf{f}_y$. Here $\mathbf{f}_x$, $\mathbf{f}_y$ are the components of F_{ext} sampled (by interpolation) at the current points, and I is the identity. The matrix $(A + \gamma I)$ is fixed, so it is inverted or factorized once; each iteration is then a cheap matrix–vector product. The implicit treatment of the stiff fourth-order term is what lets the snake take reasonable steps without blowing up. The step size is $1/\gamma$: a larger γ means smaller, safer steps.

```
import numpy as np

def snake_step_matrix(n, alpha, beta, gamma):
    """(A + gamma*I)^-1 for a closed snake with n points (h = 1)."""
    row = np.zeros(n)
    row[[0, 1, 2, -2, -1]] = [2*alpha + 6*beta, -alpha - 4*beta, beta,
                              beta, -alpha - 4*beta]
    A = np.stack([np.roll(row, i) for i in range(n)])
    return np.linalg.inv(A + gamma * np.eye(n))

M = snake_step_matrix(100, alpha=0.05, beta=0.05, gamma=1.0)
# one iteration, given external forces fx, fy sampled at the points:
# x = M @ (gamma * x + fx);  y = M @ (gamma * y + fy)
```

scikit-image's `active_contour` builds the same internal-energy matrix and offers the edge and line energies above, but it is not an exact copy of the update above: it computes the image gradient itself and caps how far a point may move per iteration [12]:

```
import numpy as np
from skimage import data
from skimage.color import rgb2gray
from skimage.filters import gaussian
from skimage.segmentation import active_contour
```

```
img = rgb2gray(data.astronaut())
s = np.linspace(0, 2 * np.pi, 400)
init = np.column_stack([100 + 100 * np.sin(s),   # rows
                        220 + 100 * np.cos(s)])  # columns
snake = active_contour(gaussian(img, sigma=3), init,
                       alpha=0.015, beta=10, gamma=0.001)
print(snake.shape)  # (400, 2): the final (row, col) points
```

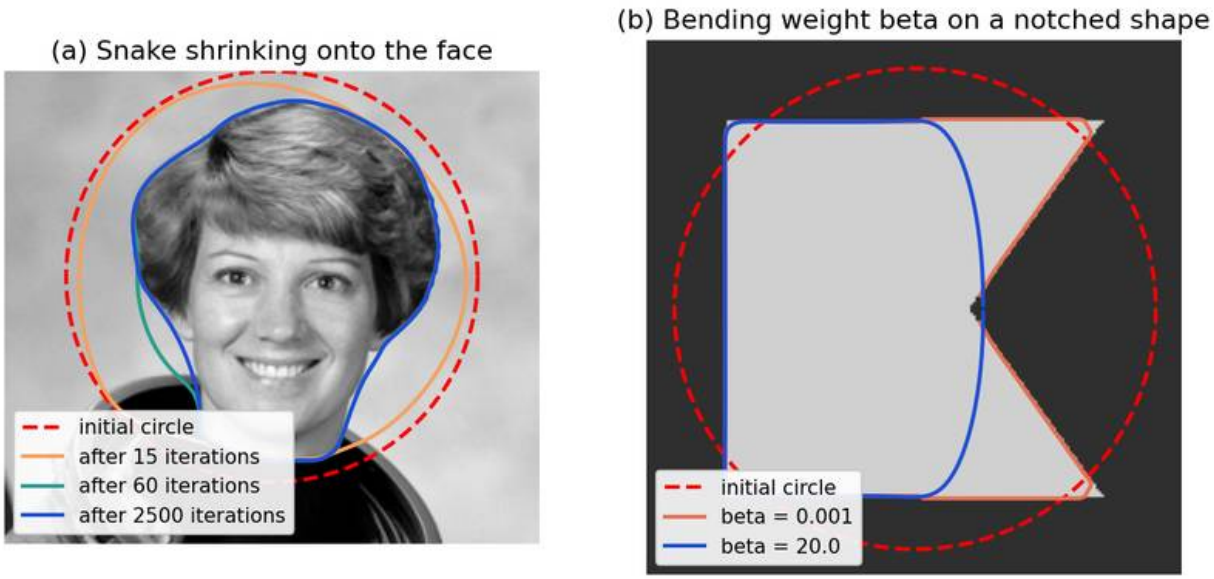


Figure 11.1 — (a) A snake started as a circle shrinks under tension until edge forces stop it on the hair and face outline. (b) The bending weight β: with $\beta = 0.001$ the snake follows the V notch and the sharp corners; with $\beta = 20$ it refuses to bend that hard and bridges the notch with a smooth arc.

Figure 11.1(a) shows the typical life of a snake: it shrinks quickly through the flat background (tension alone), then slows as it enters the zone where the blurred edge map has a slope, and finally locks onto the strongest nearby edges. Panel (b) shows the trade-off that β controls. A stiff snake is robust to noise and small gaps, but it cannot follow corners or concavities.

Choosing the external force

The external force decides *what* the snake looks for. Common choices:

- **Gradient of an edge map.** Compute an edge map $e(x, y) \geq 0$ that is large on edges, for example $e = |\nabla(G_\sigma * f)|$ (or its square), and use $F_{\text{ext}} = \kappa \nabla e$ with a weight κ. The force points uphill on e, toward the ridge of the edge.
- **Line attraction.** Use $F_{\text{ext}} = \pm\nabla(G_\sigma * f)$ to follow dark or bright curvilinear structures (roads, vessels, text strokes).

All of them share the same weakness. The gradient of an edge map is essentially zero away from edges (Figure 11.2a). A snake that starts far from the boundary feels nothing but its own tension. And inside a narrow concavity the forces from the two walls point sideways and cancel, so nothing pulls the snake *down* into it. These two problems, **limited capture range** and **poor convergence into concavities**, motivated GVF.

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

Gradient vector flow (GVF)

GVF keeps the good part of ∇e (accurate, pointing at edges where edges exist) and fills in the rest of the image by smooth diffusion. Xu and Prince [3] define the GVF field $\mathbf{w} = (u, v)$ as the smoothest vector field that stays close to ∇e wherever $|\nabla e|$ is large; a weight $\mu > 0$ balances the two and should grow with the noise level. The field is computed by iterating a diffusion equation: far from edges it simply diffuses outward from the edges, carrying their direction with it. The explicit update contains both diffusion and a reaction term pulling the field toward the edge gradient. Its time step must account for both: on a unit-spaced grid, the code uses $\Delta t = 0.9/(4\mu + \max|\nabla e|^2)$. The factor 0.9 is a conservative safety margin, not a parameter of the GVF energy.

Online only: The GVF energy and its diffusion equations — https://blog.shyandram.dev/en/learn/image-processing/active-contours/notes/the-gvf-energy-and-its-diffusion-equations/

```
import numpy as np
from scipy import ndimage as ndi

def gvf(f, mu=0.2, iters=500):
    """GVF on a unit-spaced grid, with a conservative explicit time step."""
    f = np.asarray(f, dtype=float)
    if f.ndim != 2 or min(f.shape) < 2 or not np.isfinite(f).all():
        raise ValueError("f must be a finite 2-D array with both dimensions >= 2")
    if not np.isfinite(mu) or mu <= 0:
        raise ValueError("mu must be finite and positive")
    fy, fx = np.gradient(f)
    mag2 = fx**2 + fy**2
    dt = 0.9 / (4 * mu + mag2.max())  # diffusion AND reaction enter the bound
    u, v = fx.copy(), fy.copy()
    for _ in range(iters):
        u += dt * (mu * ndi.laplace(u, mode="nearest") - (u - fx) * mag2)
        v += dt * (mu * ndi.laplace(v, mode="nearest") - (v - fy) * mag2)
    return u, v

edge_map = np.array([[0., 1.], [1., 0.]])
u, v = gvf(edge_map, mu=0.25, iters=100)
print(np.isfinite(u).all() and np.isfinite(v).all())  # True
print(np.round(u, 3))  # [[ 0.8  0.8], [-0.8 -0.8]]
```

The snake then uses $F_{\text{ext}} = \kappa\,\mathbf{w}$ in place of $\kappa\,\nabla e$, with no other change to the algorithm.

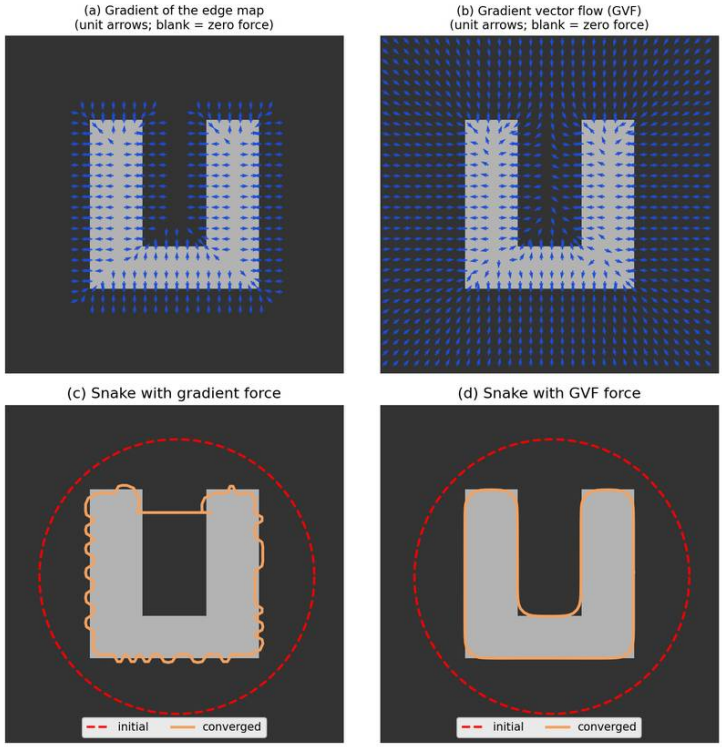


Figure 11.2 — (a) The gradient of an edge map is non-zero only in a thin band around edges. (b) GVF diffuses those vectors over the whole image; inside the notch they point downward. (c) With the gradient force, a snake started on the dashed circle stalls across the mouth of the notch (its wiggles are where it is caught on the edge band with nothing to smooth it). (d) The same snake, same parameters, driven by GVF, follows the concavity to its bottom.

Figure 11.2 reproduces the classic test of [3] on a synthetic U shape. Two things changed: the snake now feels the object from anywhere in the image (large capture range), and inside the notch the diffused vectors point down rather than sideways, so the snake is pulled into the concavity. One implementation detail matters: as the snake enters the notch its length grows, so the points must be **resampled** to a roughly constant spacing every few dozen iterations, or the chain becomes too sparse to follow the walls.

The balloon force

Cohen [4] attacked the capture-range problem from another angle. If you initialize a small snake *inside* the object, tension makes it shrink to a point before any edge force can reach it. So add a pressure:

$$F_{\text{ext}}(v) = \kappa\,\nabla e(v) \;+\; k_1\,\mathbf{n}(s),$$

where $\mathbf{n}(s)$ is the unit outward normal of the curve at $v(s)$ and k_1 is the pressure. With $k_1 > 0$

the snake inflates like a balloon; with $k_1 < 0$ it deflates. The edge force must be able to beat the pressure at a real boundary ($\kappa|\nabla e| > |k_1|$ there), but not at the weak, noisy gradients inside the object. Cohen also suggests normalizing the edge force, $\nabla e/|\nabla e|$, so that one constant controls its strength everywhere.

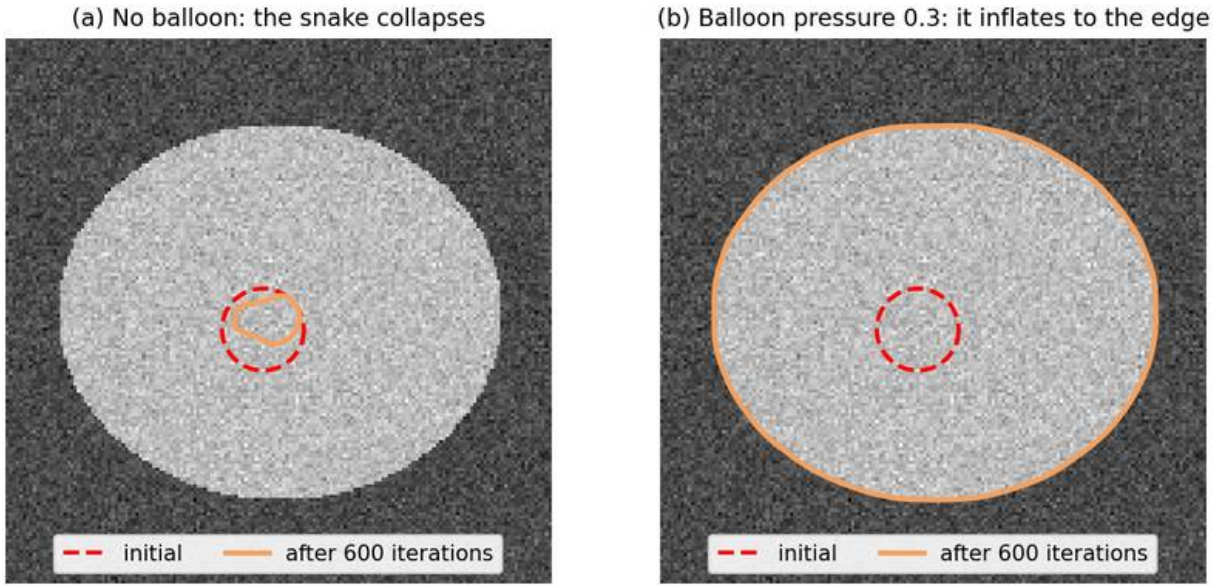


Figure 11.3 — The balloon force. (a) A small snake inside a noisy object, driven only by tension and edge forces, collapses because no edge is within reach. (b) Adding an outward pressure of 0.3 pixel per iteration inflates the snake until the edge force at the true boundary balances the pressure.

The balloon force has a cost: you must know in advance whether to inflate or deflate, and too much pressure carries the snake straight through weak edges. GVF and balloon forces are complementary and are often combined.

Limitations of snakes

- **Initialization.** The energy has many local minima. Even with GVF, a snake started across two objects or on the wrong side of a strong clutter edge will converge to the wrong answer.
- **Parameter tuning.** α, β, γ, κ, the blur σ, the pressure k_1 and the GVF μ interact. Values that work on one image often fail on the next.
- **Point management.** The points drift along the curve, bunch up in some places and spread out in others. Practical snakes resample the curve and must detect when it crosses itself.
- **Topology is fixed.** A snake is one closed chain. It cannot split into two when it encloses two objects (Figure 11.6a), and it cannot merge with another snake, without extra, error-prone bookkeeping.

The last limitation is the one level sets remove.

Segmentation using level sets

Instead of tracking the boundary itself, track a landscape whose shoreline is the boundary. Move the landscape up or down, and the shoreline moves with it. If the water rises enough to separate two hills, the one shoreline becomes two, with no special code.

The implicit representation

In plain words: store a height at every pixel; the curve is the shoreline where the height is zero.

A **level-set function** is a scalar function $\phi : \Omega \times [0, \infty) \to \mathbb{R}$ such that the curve at time t is its zero level set:

$$C(t) = \{\, (x, y) \in \Omega \;:\; \phi(x, y, t) = 0 \,\}.$$

We use the convention $\phi < 0$ inside the curve and $\phi > 0$ outside. (Some papers, including [8], use the opposite sign; the equations change sign accordingly.) Two geometric quantities come for free from ϕ:

$$\mathbf{N} = \frac{\nabla \phi}{|\nabla \phi|}, \qquad \kappa = \nabla \cdot \frac{\nabla \phi}{|\nabla \phi|},$$

where $\mathbf{N}$ is the unit **outward** normal and κ is the **curvature** of the level curve through each point (positive on convex parts; for a circle of radius r, $\kappa = 1/r$).

The most convenient ϕ is the **signed distance function**: $|\phi(x, y)|$ is the Euclidean distance from (x, y) to C, with the sign telling inside from outside. It satisfies $|\nabla \phi| = 1$ almost everywhere, which keeps the numerics well behaved.

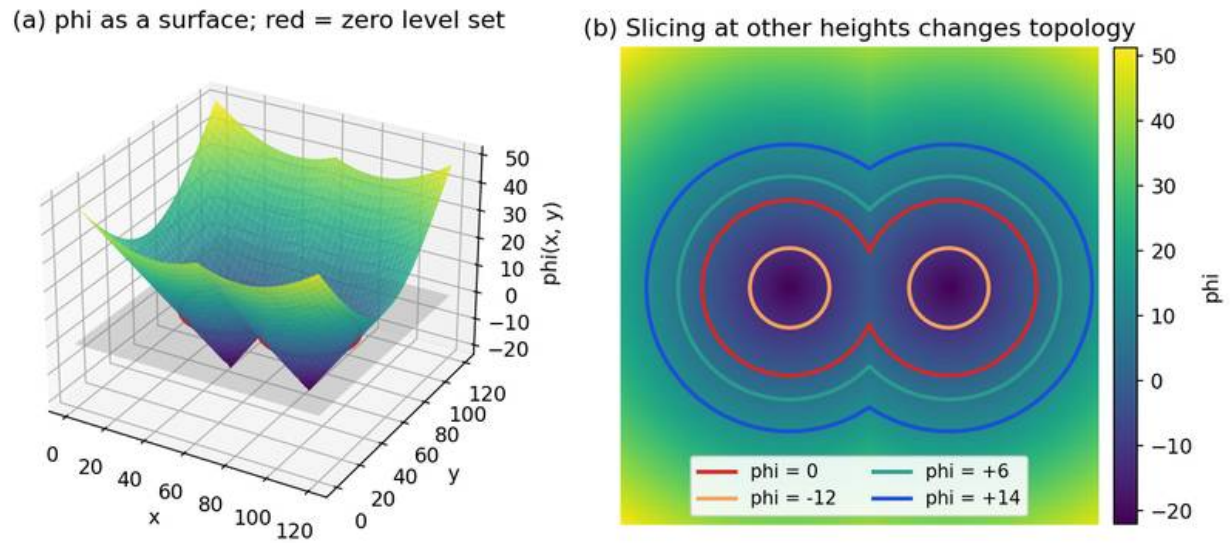


Figure 11.4 — A level-set function φ for two touching discs, built as a signed distance. (a) The curve is where the surface crosses the plane $\varphi = 0$. (b) Slicing the same surface at other heights gives two separate

curves ($\varphi = -12$) or one merged curve ($\varphi = +6, +14$). Moving φ up or down is how a level-set contour splits and merges without any special handling.

The level-set equation

Suppose each point of the curve moves along its outward normal with speed F (positive means outward):

$$\frac{\partial C}{\partial t} = F\,\mathbf{N}.$$

Points on the curve keep the value zero, so $\phi(C(t), t) = 0$ for all t. Differentiate with the chain rule:

$$\phi_t + \nabla\phi \cdot \frac{\partial C}{\partial t} = 0 \implies \phi_t + F\,\nabla\phi \cdot \frac{\nabla\phi}{|\nabla\phi|} = 0 \implies \boxed{\phi_t + F\,|\nabla\phi| = 0}$$

This is the level-set equation of Osher and Sethian [5]. ϕ_t is the time derivative of ϕ and F is the **speed function**, which may depend on position (image data), on the geometry of the curve (curvature κ), or on global statistics. The equation is solved on the fixed pixel grid for all (x, y), not only on the curve; the curve is read off at any time by extracting the zero contour. Nothing in the equation cares how many pieces the zero set has, which is why splitting and merging happen naturally.

Discrete computation: upwind differences and the CFL condition

In plain words: on a grid, the front must be updated with differences taken from the side it is coming from, the **upwind** side, as in the scheme of [5]. With ϕ_{ij}^{n} the value at pixel (i, j) after n steps of size Δt (grid spacing $\Delta x = 1$), each step subtracts $\Delta t\, F_{ij}$ times an upwind estimate of $|\nabla\phi|$ built from one-sided differences; the `evolve` function below implements it. Using a plain central difference instead produces oscillations and corners that smear or explode; upwinding is what keeps the front sharp and stable.

Online only: The upwind scheme in formulas — https://blog.shyandram.dev/en/learn/image-processing/active-contours/notes/the-upwind-scheme-in-formulas/

The explicit scheme is stable only if the front moves less than one cell per step. This is the **Courant–Friedrichs–Lewy (CFL)** condition:

$$\Delta t \max_{ij} |F_{ij}| \le c\,\Delta x, \qquad 0 < c \le 1.$$

In practice you compute Δt from the current maximum speed at each step, as the code below does.

```
import numpy as np
from scipy import ndimage as ndi

def signed_distance(mask):
```

```
    """Negative inside, positive outside, ~0 on the boundary."""
    return ndi.distance_transform_edt(~mask) - ndi.distance_transform_edt(mask)

def evolve(phi, F, steps=100, cfl=0.5):
    """Upwind scheme for phi_t + F |grad phi| = 0 (F is an array or scalar)."""
    for _ in range(steps):
        p = np.pad(phi, 1, mode="edge")
        dxm = phi - p[1:-1, :-2]; dxp = p[1:-1, 2:] - phi   # backward / forward
        dym = phi - p[:-2, 1:-1]; dyp = p[2:, 1:-1] - phi
        grad_plus = np.sqrt(np.maximum(dxm, 0)**2 + np.minimum(dxp, 0)**2 +
                            np.maximum(dym, 0)**2 + np.minimum(dyp, 0)**2)
        grad_minus = np.sqrt(np.minimum(dxm, 0)**2 + np.maximum(dxp, 0)**2 +
                             np.minimum(dym, 0)**2 + np.maximum(dyp, 0)**2)
        dt = cfl / (np.abs(F).max() + 1e-12)             # CFL: |F| dt <= dx
        phi = phi - dt * (np.maximum(F, 0) * grad_plus + np.minimum(F, 0) * grad_minus)
    return phi

yy, xx = np.mgrid[:128, :128]
phi = signed_distance((xx - 64)**2 + (yy - 64)**2 < 20**2)
phi = evolve(phi, F=1.0, steps=20)                # grow outward by 20 * 0.5 = 10 px
phi = signed_distance(phi < 0)                     # re-initialize
print(np.sqrt((phi < 0).sum() / np.pi))            # radius, about 30
```

Specifying the speed function

All the image analysis lives in F. Two families dominate.

Edge-based speed: geodesic active contours

In plain words: make the front fast in flat regions and slow on edges, so that it stops where the edges are.

Define an **edge-stopping function** that is close to 1 in flat regions and close to 0 on edges, for example

$$g(x,y) = \frac{1}{1 + |\nabla (G_\sigma * f)(x,y)|^2}.$$

Caselles, Kimmel and Sapiro [6] showed that a snake with $\beta = 0$ is closely related to finding a curve of minimal **weighted length**

$$L_g(C) = \oint_C g(C(s))\, ds,$$

where ds is arc length. In the metric defined by g, edges are "cheap" places to be, so a minimizer is a geodesic (shortest path) that hugs edges. This is the **geodesic active contour** (**GAC**). Gradient descent on L_g, plus an optional constant pressure c like Cohen's balloon, gives in our sign convention

$$\phi_t = g\,|\nabla \phi|\,(\kappa - c) + \nabla g \cdot \nabla \phi.$$

Each term has a job. $g\,\kappa\,|\nabla\phi|$ is curvature flow (shortening and smoothing) that slows to a stop near edges. $-c\,g\,|\nabla\phi|$ is the pressure: $c > 0$ inflates, $c < 0$ deflates, again switched off on edges. $\nabla g \cdot \nabla\phi$ is an advection term that pulls the curve into the bottom of the edge valley, where g is minimal, and stops it from overshooting. Unlike a snake, the GAC is **intrinsic**: it does not depend on how the curve is parametrized, and in level-set form it splits and merges freely.

Because g never reaches exactly zero, a strong pressure still leaks through weak or blurred edges. That is the motivation for region-based speeds.

Region-based speed: Mumford–Shah and Chan–Vese

In plain words: instead of looking for edges, choose the curve that splits the image into an inside and an outside that are each as uniform as possible.

Mumford and Shah [7] posed segmentation as finding a piecewise-smooth approximation of the image together with a set of boundaries. Their energy balances three wishes: stay close to the image, be smooth everywhere except across the boundaries, and keep the boundaries short. This energy is hard to minimize in general.

Online only: The Mumford–Shah energy — https://blog.shyandram.dev/en/learn/image-processing/active-contours/notes/the-mumfordshah-energy/

Chan and Vese [8] restricted u to be **piecewise constant with two values**: c_1 inside C and c_2 outside. The energy becomes

$$E_{\mathrm{CV}}(c_1, c_2, C) = \mu\,|C| + \lambda_1 \int_{\mathrm{inside}(C)} (f - c_1)^2\,dx\,dy + \lambda_2 \int_{\mathrm{outside}(C)} (f - c_2)^2\,dx\,dy,$$

(the original also has an optional area term $\nu\,\mathrm{Area}(\mathrm{inside}(C))$, usually set to zero). $\mu \geq 0$ weights smoothness and $\lambda_1, \lambda_2 > 0$ weight how well each region is fitted by its constant. Note what is missing: there is no gradient. The model can segment objects whose boundaries are blurred or defined only by a change of average intensity, which is why [8] called it "active contours without edges".

To write this in level-set form, use the Heaviside function $H(z)$ (1 for $z \geq 0$, 0 otherwise) and its derivative, the Dirac delta $\delta(z)$. With $\phi < 0$ inside, the inside indicator is $H(-\phi)$ and the length is $|C| = \int \delta(\phi)|\nabla\phi|$. For fixed ϕ, the optimal constants are the region means

$$c_1 = \frac{\int f\,H(-\phi)}{\int H(-\phi)}, \qquad c_2 = \frac{\int f\,H(\phi)}{\int H(\phi)}.$$

For fixed c_1, c_2, gradient descent in ϕ gives

$$\phi_t = \delta_\varepsilon(\phi)\,\big[\,\mu\,\kappa + \lambda_1 (f - c_1)^2 - \lambda_2 (f - c_2)^2\,\big],$$

where δ_ε is a smoothed delta of width ε. Read it pixel by pixel: if a pixel's intensity is far from c_1 but close to c_2, the bracket is positive, ϕ rises, and the pixel moves outside. The curvature term keeps the boundary short. The algorithm alternates between updating the means and updating ϕ. Because δ_ε is non-zero away from the zero level set, the model can also **create new contours** far from the initial one, which helps detect interior holes and disconnected objects in the examples of

[8]. This behavior depends on the chosen regularization; the nonconvex energy can still converge to different local minima from different initial curves.

Online only: Chan–Vese: locate a boundary by fitting the regions on both sides — https://blog.shyandram.dev/en/learn/image-processing/active-contours/papers/chanvese-locate-a-boundary-by-fitting-the-regions-on-both-sides/

Chan–Vese is one member of a larger family of region-based models: any probability model of the regions can take the place of the two constants [14].

Online only: Other region models — https://blog.shyandram.dev/en/learn/image-processing/active-contours/notes/other-region-models/

```
from skimage import data, img_as_float
from skimage.segmentation import chan_vese, morphological_chan_vese

cam = img_as_float(data.camera())
seg, phi, energies = chan_vese(cam, mu=0.25, lambda1=1, lambda2=1,
                               dt=0.5, max_num_iter=200, extended_output=True)
fast = morphological_chan_vese(cam, num_iter=100, init_level_set="checkerboard",
                               smoothing=3)
```

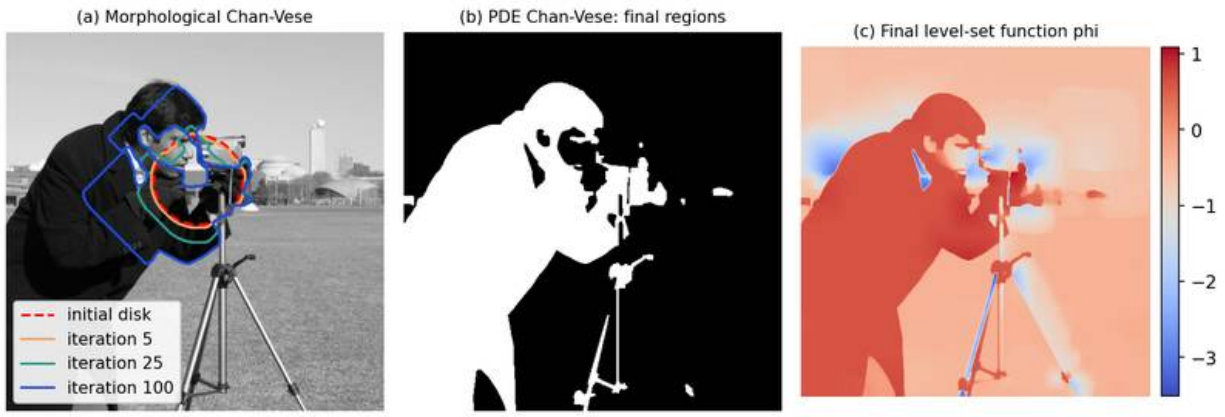


Figure 11.5 — Chan–Vese on the cameraman image. (a) The morphological approximation started from a disk on the face; the region grows into everything that is closer to the dark mean than to the bright mean. (b) The PDE version started from a checkerboard: the two-phase split separates the dark figure, camera and tripod from sky and grass, and also produces new, disconnected contours (tripod legs, distant buildings) that were never connected to the initial curve. (c) The final φ; note that scikit-image uses the opposite sign convention ($\varphi > 0$ inside) [12].

Initializing and reinitializing as a signed distance function

The initial ϕ is usually the signed distance to a user-drawn curve, a box, a disk, or many small circles (a "checkerboard") so that the evolution starts near every possible object. As the PDE

runs, ϕ drifts away from a distance function: it becomes very steep in some places and very flat in others. Steep regions force tiny time steps; flat regions make the zero crossing poorly located and sensitive to noise.

The classical remedy is **reinitialization**: every few iterations, replace ϕ by the signed distance to its current zero level set without moving that zero set, for example with a fast Euclidean distance transform as in the code above. Li, Xu, Gui and Fox [10] proposed **distance-regularized level-set evolution** (**DRLSE**), which adds an energy penalty that keeps $|\nabla\phi|$ close to 1 and so removes the need for reinitialization.

Online only: Reinitialization and DRLSE in detail — https://blog.shyandram.dev/en/learn/image-processing/active-contours/notes/reinitialization-and-drlse-in-detail/

Improving efficiency

A full-grid level-set solver updates every pixel at every step, although only the zero crossing matters. Three standard speed-ups:

- **Narrow band.** Adalsteinsson and Sethian [9] update ϕ only within a band of half-width w (a few pixels) around the zero level set. When the front approaches the edge of the band, the band is rebuilt around the new front (with a reinitialization). The work per step drops from the image area to roughly the curve length times w.
- **Larger stable steps.** Semi-implicit or operator-splitting schemes allow larger Δt for the stiff curvature term, at the cost of solving a linear system per step.
- **Morphological approximations.** Márquez-Neila, Baumela and Álvarez [11] replace the PDE by a sequence of binary morphological operators (dilations, erosions and smoothing operators) acting on a *binary* level set. The result is fast, has no time step, needs no reinitialization and never becomes unstable.

scikit-image provides `morphological_geodesic_active_contour` and `morphological_chan_vese` based on [11], alongside the PDE-based `chan_vese` [12]. Keep in mind that the morphological variants **approximate** the PDEs. They move on the pixel grid, their "smoothing" parameter is a number of operator applications rather than the weight μ, and their results are close to, not identical with, the continuous models.

Online only: How the morphological approximation works — https://blog.shyandram.dev/en/learn/image-processing/active-contours/notes/how-the-morphological-approximation-works/

```
from skimage import data, img_as_float
from skimage.segmentation import (inverse_gaussian_gradient, disk_level_set,
                                  morphological_geodesic_active_contour)

coins = img_as_float(data.coins())
g = inverse_gaussian_gradient(coins, alpha=100, sigma=2)   # ~0 on edges, ~1 elsewhere
init = disk_level_set(coins.shape, radius=150)
mask = morphological_geodesic_active_contour(g, num_iter=250, init_level_set=init,
                                             smoothing=1, balloon=-1, threshold=0.69)
```

Here `inverse_gaussian_gradient` builds the edge-stopping function g, `balloon=-1` shrinks the initial disk (the role of $c < 0$), and `threshold` decides which values of g count as edges that stop the balloon.

Snakes vs level sets in practice

Both families minimize similar energies. The difference is mostly in representation, and the representation decides what is easy.

In short: a snake has few unknowns (its n points), is cheap per iteration, is easy to edit interactively and can carry point labels, but its topology is fixed and its points drift and can cross. A level set has one unknown per pixel (or per band pixel), but it splits and merges automatically, cannot self-intersect and extends directly to 3D. Their typical failures differ too: snakes stall on clutter or in concavities, edge-based level sets leak through weak edges, and region-based ones merge similar-looking regions.

Online only: Snakes vs level sets, point by point — https://blog.shyandram.dev/en/learn/image-processing/active-contours/notes/snakes-vs-level-sets-point-by-point/

Xu, Yezzi and Prince studied the relationship between the two formulations [3]; for many energies the choice is about implementation, not about what can be modeled in principle. A rule of thumb:

- Use a **snake** when you need one boundary of known topology, interactively, fast, or with point correspondences (tracking a contour across video frames, fitting a lip outline, delineating a building footprint).
- Use a **level set** when the number of objects is unknown, objects may touch or split, the boundary is ill-defined by gradients (region-based models), or you work in 3D.

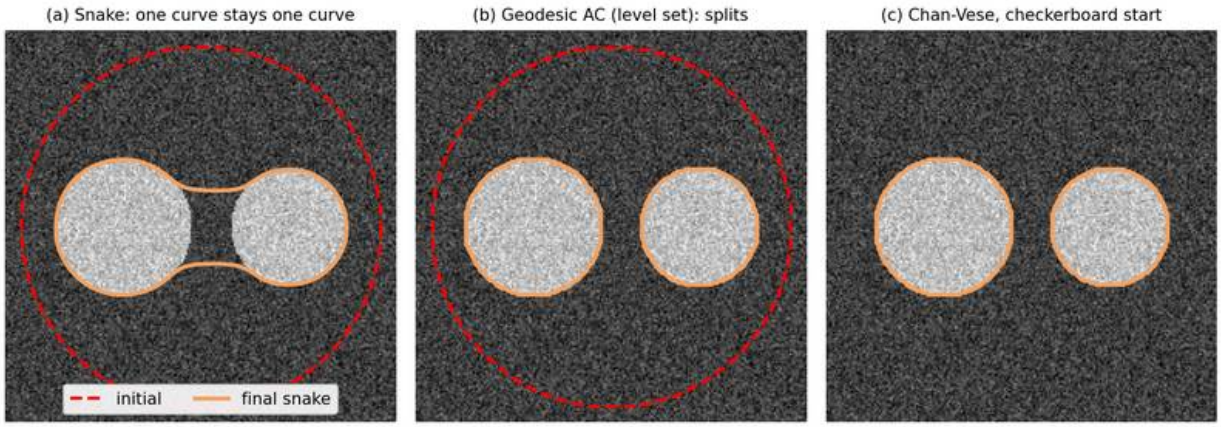


Figure 11.6 — Topology in practice. (a) A snake started around both discs can only shrink into a single curve that wraps both and bridges the gap. (b) A geodesic active contour (morphological level-set version) from the same initial circle splits into two curves when its two halves meet in the middle. (c) Chan–Vese started from a checkerboard does not need a meaningful initial curve at all.

Modern view

Active contours are no longer the default way to segment an image; deep networks are. But the ideas of this chapter, energies that combine boundary length with region fit, explicit contours deformed iteratively, and level sets as a shape representation, are still active ingredients in deep segmentation. Six papers give a good map.

Deformable models in medical imaging. McInerney and Terzopoulos [13] wrote the standard early survey. Its main takeaway still holds: deformable models are most useful when the image alone is ambiguous and the model must contribute smoothness and prior knowledge.

Statistical region-based level sets. Cremers, Rousson and Deriche [14] derive region-based level-set methods from one Bayesian framework in which each region gets a probability model (intensity, color, texture or motion). They argue that region-based energies are less sensitive to noise and have fewer local minima than edge-based snakes.

A recent taxonomy. Chen, Ge, Wang and Chen [15] sort active contour models into edge-based, region-based and hybrid families and compare them with deep segmentation. Classical models need no training data and give smooth closed boundaries but are sensitive to initialization and parameters; deep models are more accurate where labelled data exist; the two are complementary.

Shape constraints in deep segmentation. Bohlender, Oksuz and Mukhopadhyay [16] survey how shape knowledge, including active contours and signed distance maps, is injected into deep medical image segmentation to fix the fragmented regions and topological errors of pixel-wise classifiers.

Online only: The four surveys in more detail — https://blog.shyandram.dev/en/learn/image-processing/active-contours/papers/the-four-surveys-in-more-detail/

Explicit contours learned end to end. Deep Snake by Peng et al. [17] keeps the snake's representation, a closed polygon deformed iteratively, but learns the update: a circular convolution over CNN features sampled at the vertices predicts per-vertex offsets toward the boundary, and the authors report real-time speed for instance segmentation. The circular convolution plays the role of the classical snake's pentadiagonal matrix A: it couples each vertex to its neighbours along the closed curve.

Active-contour energies as loss functions. Instead of running a contour at test time, several methods put the energy into the training loss: a supervised Chan–Vese-like loss with length and region terms [18], a label-free Mumford–Shah-style loss on a network's softmax outputs that also supports unsupervised and semi-supervised training [19], and the **boundary loss** [20], a regional integral weighted by the signed distance map of the ground truth, designed for highly unbalanced problems such as small lesions.

Online only: Three active-contour losses in detail — https://blog.shyandram.dev/en/learn/image-processing/active-contours/notes/three-active-contour-losses-in-detail/

What deep learning changed, and what it did not:

- **Changed: the data term.** Hand-made forces (∇e, GVF, g, region means) are replaced by learned features. This is the main reason accuracy improved.

- **Not changed: the regularity prior.** Boundary length, curvature penalties, signed distance representations and region homogeneity reappear as loss terms and output representations, because pixel classifiers still produce ragged, fragmented boundaries without them.
- **Not changed: the explicit vs implicit trade-off.** Contour-based networks inherit the snake's fixed topology (one polygon per instance) and its speed; mask and distance-map networks inherit the level set's flexibility.
- **Still useful on its own.** When there are no labelled data, a Chan–Vese or GAC run is a few lines of scikit-image and needs no training, which makes it a good baseline, a pseudo-label generator, or a post-processing step that cleans up a network's mask.

Key takeaways

- An active contour segments by **minimizing an energy over curves**: a regularity term (length, bending) plus a data term (edges or region fit). Gradient descent on that energy is a PDE in artificial time.
- A **snake** is an explicit chain of points. Its Euler–Lagrange equation $\alpha v_{ss} - \beta v_{ssss} + F_{\text{ext}} = 0$ is solved with a semi-implicit step $\mathbf{x}^t = (A + \gamma I)^{-1}(\gamma \mathbf{x}^{t-1} + \mathbf{f}_x)$.
- Plain gradient forces have a **small capture range** and fail in **concavities**. GVF diffuses edge vectors across the image; the balloon force inflates or deflates the curve.
- A **level set** stores the curve as $\{\phi = 0\}$ and evolves $\phi_t + F|\nabla\phi| = 0$ with **upwind differences** under a **CFL** step limit. Topology changes are automatic.
- The speed F carries the model: **geodesic active contours** minimize edge-weighted length; **Chan–Vese** fits two region means and works without edges.
- Keep ϕ close to a **signed distance** (reinitialization or DRLSE), and save work with **narrow bands** or **morphological** approximations, which approximate rather than solve the PDEs.
- In deep learning the same ideas survive as **learned contour deformation** (Deep Snake) and as **loss functions** built from length, region and signed-distance terms.

Exercises

1. Tension alone. A closed snake with $\beta = 0$ and no external force is initialized as a circle of radius R sampled at n evenly spaced points. Using the discrete internal force $\alpha(v_{i+1} - 2v_i + v_{i-1})$, show that the circle stays a circle and find how its radius changes in one explicit step of size τ.

> **Hint**
>
> For a regular polygon with angle step $\theta = 2\pi/n$, $v_{i+1} + v_{i-1} = 2\cos\theta\, v_i$ (measured from the centre). The force is $-2\alpha(1 - \cos\theta)\, v_i$, pointing to the centre, so $R \mapsto R\,[1 - 2\alpha\tau(1 - \cos\theta)]$. The circle shrinks geometrically; note that the rate depends on n, which is one reason snake parameters are tied to the sampling.

2. Concavity test. Using the `gvf` function above, build the U shape of Figure 11.2 and vary the GVF weight μ (0.05, 0.1, 0.2) and the number of GVF iterations. Which parameter controls how far the field reaches from the edges, and which controls how smooth it is?

Hint

The diffusion length grows roughly with $\sqrt{\mu n \Delta t}$, where n is the iteration count. A larger μ also changes the smoothness/data-fit balance. Because the code adapts Δt to both diffusion and reaction, $\mu > 0.25$ is not automatically unstable; compare equal evolution time or a fixed convergence tolerance when changing μ.

3. Signs and directions. With our convention ($\phi < 0$ inside), what happens to a circle under $\phi_t + F|\nabla\phi| = 0$ when $F = -1$? When $F = -\kappa$? Write the radius $r(t)$ in each case.

Answer

$F = -1$: the circle shrinks at unit speed, $r(t) = r_0 - t$. $F = -\kappa = -1/r$: curve-shortening flow, $\dot{r} = -1/r$, so $r(t) = \sqrt{r_0^2 - 2t}$, and the circle vanishes at $t = r_0^2/2$.

4. CFL in practice. Run the `evolve` snippet with `cfl=0.5`, `1.0` and `2.0` on the same circle. Measure the final radius and look at the zero contour. What breaks, and why does the narrow band not remove this limit?

Hint

Above 1 the front tries to cross more than one cell per step; upwind information comes from the wrong place and the contour becomes jagged or the values blow up. The narrow band reduces the number of pixels updated, not the size of each step.

5. When Chan–Vese fails. Create a synthetic image whose left half is a smooth gradient from black to white and whose right half contains a gray disc on a gray background of the same mean but different variance (add strong noise only inside the disc). Run `chan_vese`. Explain the result from the energy, and propose a change of the region model that would succeed.

Hint

Chan–Vese only compares means, so the disc (same mean) is invisible and the ramp is cut in the middle. Modeling each region by a Gaussian with its own mean *and* variance, as in the statistical framework of [14], makes the disc separable.

6. Energy as a loss. Write the Chan–Vese energy for a soft mask $m(x, y) \in [0, 1]$ (inside probability) instead of a level set, with the length term replaced by the total variation $\int |\nabla m|$. Show that for fixed m the optimal c_1, c_2 are weighted means, and explain why this function can be used to train a network without labels.

Hint

$E = \mu \int |\nabla m| + \lambda_1 \int m\,(f - c_1)^2 + \lambda_2 \int (1 - m)(f - c_2)^2$. Setting $\partial E / \partial c_1 = 0$ gives $c_1 = \int mf / \int m$. Nothing in E refers to ground truth, only to the image itself, so gradients with respect to the network producing m exist without labels; compare with [19].

References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018, Ch. 11. publisher page
2. M. Kass, A. Witkin, and D. Terzopoulos, “Snakes: Active contour models,” *International Journal of Computer Vision*, vol. 1, no. 4, pp. 321–331, 1988. doi:10.1007/BF00133570
3. C. Xu and J. L. Prince, “Snakes, shapes, and gradient vector flow,” *IEEE Transactions on Image Processing*, vol. 7, no. 3, pp. 359–369, 1998; see also C. Xu, A. Yezzi Jr., and J. L. Prince, “On the relationship between parametric and geometric active contours,” *Proc. 34th Asilomar Conf. on Signals, Systems, and Computers*, 2000. GVF project page with both papers
4. L. D. Cohen, “On active contour models and balloons,” *CVGIP: Image Understanding*, vol. 53, no. 2, pp. 211–218, 1991. author PDF
5. S. Osher and J. A. Sethian, “Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton–Jacobi formulations,” *Journal of Computational Physics*, vol. 79, pp. 12–49, 1988. doi:10.1016/0021-9991(88)90002-2
6. V. Caselles, R. Kimmel, and G. Sapiro, “Geodesic active contours,” *International Journal of Computer Vision*, vol. 22, no. 1, pp. 61–79, 1997. PDF
7. D. Mumford and J. Shah, “Optimal approximations by piecewise smooth functions and associated variational problems,” *Communications on Pure and Applied Mathematics*, vol. 42, no. 5, pp. 577–685, 1989. doi:10.1002/cpa.3160420503
8. T. F. Chan and L. A. Vese, “Active contours without edges,” *IEEE Transactions on Image Processing*, vol. 10, no. 2, pp. 266–277, 2001. doi:10.1109/83.902291
9. D. Adalsteinsson and J. A. Sethian, “A fast level set method for propagating interfaces,” *Journal of Computational Physics*, vol. 118, pp. 269–277, 1995. summary page
10. C. Li, C. Xu, C. Gui, and M. D. Fox, “Distance regularized level set evolution and its application to image segmentation,” *IEEE Transactions on Image Processing*, vol. 19, no. 12, pp. 3243–3254, 2010. doi:10.1109/TIP.2010.2069690
11. P. Márquez-Neila, L. Baumela, and L. Álvarez, “A morphological approach to curvature-based evolution of curves and surfaces,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014. doi:10.1109/TPAMI.2013.106
12. scikit-image developers, “skimage.segmentation” API reference (`active_contour`, `chan_vese`, `morphological_chan_vese`, `morphological_geodesic_active_contour`). docs
13. T. McInerney and D. Terzopoulos, “Deformable models in medical image analysis: A survey,” *Medical Image Analysis*, vol. 1, no. 2, 1996. author PDF
14. D. Cremers, M. Rousson, and R. Deriche, “A review of statistical approaches to level set segmentation: Integrating color, texture, motion and shape,” *International Journal of Computer Vision*, vol. 72, no. 2, pp. 195–215, 2007. doi:10.1007/s11263-006-8711-1
15. Y. Chen, P. Ge, G. Wang, and H. Chen, “An overview of intelligent image segmentation using active contour models,” *Intelligence & Robotics*, vol. 3, no. 1, pp. 23–55, 2023. open access
16. S. Bohlender, I. Oksuz, and A. Mukhopadhyay, “A survey on shape-constraint deep learning for medical image segmentation,” *IEEE Reviews in Biomedical Engineering*, vol. 16, pp. 225–240, 2023. arXiv:2101.07721
17. S. Peng, W. Jiang, H. Pi, X. Li, H. Bao, and X. Zhou, “Deep snake for real-time instance segmentation,” *CVPR*, 2020. arXiv:2001.01629

18. X. Chen, B. M. Williams, S. R. Vallabhaneni, G. Czanner, R. Williams, and Y. Zheng, “Learning active contour models for medical image segmentation,” *CVPR*, 2019. CVF open access
19. B. Kim and J. C. Ye, “Mumford–Shah loss functional for image segmentation with deep learning,” *IEEE Transactions on Image Processing*, vol. 29, pp. 1856–1866, 2020. doi ·arXiv:1904.02872
20. H. Kervadec, J. Bouchtiba, C. Desrosiers, E. Granger, J. Dolz, and I. Ben Ayed, “Boundary loss for highly unbalanced segmentation,” *Medical Image Analysis*, vol. 67, 2021 (first presented at MIDL 2019). arXiv:1812.07032

IP 12 • Feature Extraction

Turn regions, boundaries and whole images into compact numbers that stay the same when the object moves, turns, grows or changes brightness, from chain codes and GLCM texture to Harris, MSER and SIFT.

Online edition: https://blog.shyandram.dev/en/learn/image-processing/feature-extraction/

What you'll learn

- The difference between a *feature* and a *descriptor*, and the four invariances most descriptors try to have: translation, rotation, scale and illumination.
- How to turn a segmented boundary into something measurable: chain codes, polygons, signatures and skeletons, and then into numbers such as shape numbers and Fourier descriptors.
- How to describe whole regions by size, shape, topology (Euler number), texture (histogram moments, co-occurrence matrices, spectra) and moment invariants.
- How principal component analysis (PCA) aligns objects and compresses many measurements into a few.
- How the Harris–Stephens corner detector, maximally stable extremal regions (MSER) and SIFT find and describe interest points that can be matched across two photos.
- How surveys of the last two decades judge these handcrafted features, and where learned features (SuperPoint, LoFTR, LightGlue, DINOv2) have replaced them.

The big picture

Segmentation (IP 10 and IP 11) told us *where* objects are. A program still cannot "look at" a region the way we do. It needs a short list of numbers that says *what the region is like*, so that it can compare, sort, search and recognize. Producing that list is feature extraction.

> **Explain like I'm five**
>
> Imagine describing a toy to a friend over the phone. You cannot send a picture, so you say: "It is round, about as big as an orange, has one hole in the middle, and its surface is bumpy." Your friend can now pick the right toy out of a box, even if it is upside down, farther away, or in a darker room. Each thing you said is a **feature**. Good features are the ones that stay true no matter how the toy is turned, moved, or lit.

This chapter follows the order of Gonzalez and Woods [1]. We start with boundaries, move to regions, then to statistics over many measurements (PCA), and end with features that need no segmentation at all: corners, stable regions and SIFT keypoints. These last ones connect classical image processing to modern computer vision tasks such as panorama stitching, 3D reconstruction and visual localization.

Background: features, descriptors and invariance

Plain version. A feature is something interesting you found in an image (a corner, a blob, a region). A descriptor is the list of numbers you use to describe it. We want descriptors that do not change when things that do not matter change.

Precise version. Following [1], we separate two steps:

- **Feature detection** finds *where* a feature is: a boundary, a region, a point with its scale and orientation.
- **Feature description** assigns a vector $\mathbf{d} \in \mathbb{R}^n$ to each detected feature.

A descriptor $\mathbf{d}(\cdot)$ is **invariant** to a family of transformations $\mathcal{T}$ if

$$\mathbf{d}(T(f)) = \mathbf{d}(f) \quad \text{for all } T \in \mathcal{T},$$

where f is the image (or region) and T a transformation applied to it. It is **covariant** if the detected quantity changes *with* the transformation in a predictable way (for example, a keypoint's position moves with the object, and its detected scale doubles when the object doubles). Detectors should be covariant; descriptors should be invariant.

The transformations we care about most are:

Change	What it does to the image	Typical cure
Translation	shifts coordinates	subtract a centroid, or use differences
Rotation	rotates coordinates	use a canonical orientation, magnitudes, or rotation-free quantities
Scale	multiplies coordinates	normalize by size, or search over scale
Illumination	changes intensities, roughly $f \mapsto af + b$	use gradients (remove b) and normalize (remove a)

There is always a trade-off between invariance and *discriminative power*. A descriptor invariant to everything describes nothing: a "6" rotated by 180° is a "9". Choosing features is choosing which differences matter for your task.

Boundary preprocessing

Plain version. Before we can measure a boundary, we must list its pixels in order and, often, simplify it so that noise and pixel staircases do not dominate.

Boundary following

A segmented region is a set of pixels, but most boundary descriptors need the boundary as an *ordered*, closed sequence of points $(x_0, y_0), \dots, (x_{K-1}, y_{K-1})$. A boundary-following algorithm pro-

duces that list by walking around the region, pixel by pixel, from a fixed starting pixel until it returns to the start [1]. OpenCV's `cv2.findContours` and scikit-image's `measure.find_contours` give you this list directly.

Online only: The Moore boundary-following procedure — https://blog.shyandram.dev/en/learn/image-processing/feature-extraction/notes/the-moore-boundary-following-procedure/

Chain codes

A **chain code** replaces the list of points with a list of *directions*. Each step from one boundary pixel to the next is one of 4 or 8 directions, coded 0–3 or 0–7 counterclockwise from east [1]. The code is compact (3 bits per step for 8 directions) and already independent of translation.

Two problems remain. The code depends on the **start point**, and it depends on **rotation**. The fixes are simple:

- **Start point.** Treat the code as circular and choose the rotation of the sequence that forms the smallest integer.
- **Rotation (by multiples of 45°).** Use the **first difference**: the number of direction changes, counted counterclockwise, between consecutive elements, $d_k = (c_k - c_{k-1}) \bmod 8$, where c_k is the k-th code element. Turning the object by 90° adds 2 to every c_k and leaves every d_k unchanged.

Raw chain codes on the full pixel grid are long and noisy. In practice the boundary is first resampled on a coarser grid, so that each link spans several pixels (Figure 12.1, left).

```
import numpy as np, cv2
from skimage import data

mask = (data.horse() == 0).astype(np.uint8)          # True inside the horse
cnts, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
pts = max(cnts, key=len)[:, 0, :]                    # (x, y), 8-connected boundary

# Freeman 8-direction code: 0 = east, counted counter-clockwise (y points down)
DIRS = {(1, 0): 0, (1, -1): 1, (0, -1): 2, (-1, -1): 3,
        (-1, 0): 4, (-1, 1): 5, (0, 1): 6, (1, 1): 7}
steps = np.diff(np.vstack([pts, pts[:1]]), axis=0)
code = np.array([DIRS[tuple(s)] for s in steps])

diff = (code - np.roll(code, 1)) % 8                 # first difference: rotation invariant
shifts = [tuple(np.roll(diff, -i)) for i in range(len(diff))]
shape_number = min(shifts)                           # start-point invariant
print(len(code), code[:12], diff[:12])
```

Online only: Slope chain codes — https://blog.shyandram.dev/en/learn/image-processing/feature-extraction/notes/slope-chain-codes/

Polygonal approximation

A polygon can capture the essence of a boundary with a few vertices.

Minimum-perimeter polygons (MPP). Picture a rubber band that shrinks inside a thin strip of cells covering the boundary; the shortest closed path it settles into is the MPP [1]. The cell size controls the detail: big cells give a coarse polygon.

Online only: Minimum-perimeter polygons in more detail — https://blog.shyandram.dev/en/learn/image-processing/feature-extraction/notes/minimum-perimeter-polygons-in-more-detail/

Merging. Walk along the boundary and keep adding points to the current segment while the least-squares line-fit error stays below a threshold T. When the error exceeds T, start a new segment. This is simple, but vertices may not land at true corners.

Splitting. Join the two most distant boundary points with a chord. Find the boundary point farthest from that chord; if its distance exceeds T, make it a vertex and split the problem in two. Repeat recursively. Splitting tends to place vertices at real inflection points. `skimage.measure.approximate_polygon` implements this recursive splitting idea; Figure 12.1 (second panel) shows two tolerances.

Signatures

A **signature** reduces a 2-D boundary to a 1-D function. The simplest is the distance from the centroid to the boundary as a function of angle, $r(\theta)$, or of normalized arc length, $r(s)$ [1]. A circle gives a constant; a square gives four identical bumps. Signatures are translation invariant by construction. Rotation becomes a *circular shift* of the function, and scale becomes a multiplication, so we normalize by choosing a canonical start (for example the farthest point) and dividing by the maximum or by the standard deviation. $r(\theta)$ is a function only for star-shaped regions; for shapes like the horse in Figure 12.1, a ray from the centroid can cross the boundary more than once, which is why $r(s)$ is used there.

Skeletons and the medial axis

Plain version. Light a fire on the entire edge of a grass field at the same moment. The places where fire fronts from different sides meet form the **skeleton**.

Precise version. The **medial axis** of a region is the set of points that have more than one closest point on the region's border [1]. In practice skeletons are computed by **thinning** (see IP 09), and because they are very sensitive to small bumps on the boundary, smoothing or pruning short branches is usually needed.

Online only: The medial axis transform — https://blog.shyandram.dev/en/learn/image-processing/feature-extraction/notes/the-medial-axis-transform/

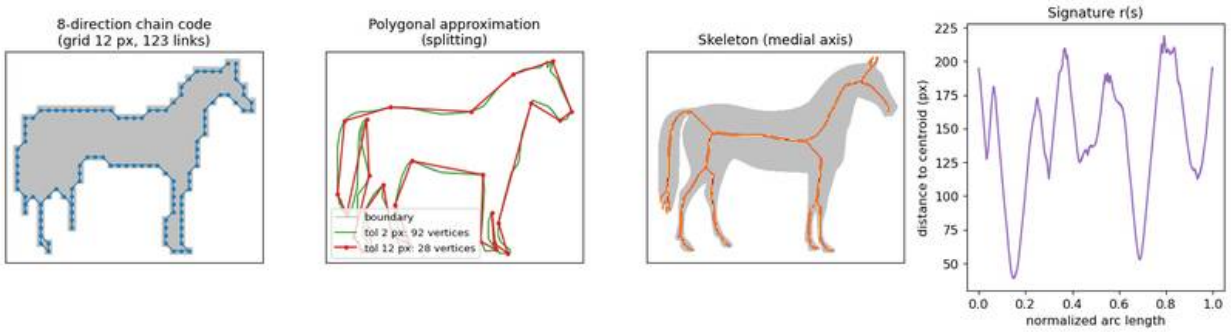


Figure 12.1 — Boundary representations of the same horse silhouette (skimage.data.horse). Left to right: an 8-direction chain code on a 12-pixel grid; polygonal approximation by recursive splitting with tolerances of 2 and 12 pixels; the skeleton obtained by thinning; and the signature $r(s)$, distance to the centroid versus normalized arc length (the deep dips are the legs).

Boundary feature descriptors

Plain version. Once a boundary is an ordered list, we can measure it: how long it is, how wide, how bent, and how it looks when written as a sum of waves.

Basic descriptors

- **Length.** The number of pixels on the boundary is a rough estimate. With an 8-connected chain code, count horizontal and vertical steps as 1 and diagonal steps as $\sqrt{2}$.
- **Diameter.** $\mathrm{Diam}(B) = \max_{i,j} D(p_i, p_j)$, where p_i, p_j are boundary points and D is a distance (usually Euclidean). The segment joining the two maximizing points is the **major axis**; the **minor axis** is perpendicular to it, with length chosen so that a box with these two sides encloses the boundary. That box is the **basic rectangle**.
- **Eccentricity.** The ratio of major-axis length to minor-axis length. (Region-based eccentricity, below, uses second moments instead.)
- **Curvature.** The rate of change of slope along the boundary. On a digital boundary it is noisy, so we usually compute the slope difference between line segments fitted on either side of a point. Where curvature changes sign, the boundary has an inflection; positive curvature (in the walking direction) marks a convex part, negative a concave part.

Shape numbers

The **shape number** of a boundary is the first difference of its 4-direction chain code, circularly shifted to form the smallest integer [1]. Its length is the **order** n of the shape. Before two shapes are compared, the grid is aligned with each shape's orientation; they are then compared by their **degree of similarity**, the largest order for which their shape numbers still agree.

Online only: Computing shape numbers of a given order — https://blog.shyandram.dev/en/learn/image-processing/feature-extraction/notes/computing-shape-numbers-of-a-given-order/

Fourier descriptors

Plain version. Walk around the boundary and write each point as a complex number. The sequence repeats every lap, so we can decompose it into waves. The first few, slow waves give the rough outline; fast waves add the details.

Precise version. Let the boundary be K points (x_k, y_k), $k = 0, \dots, K-1$, and write each as

$$s(k) = x(k) + j\,y(k).$$

The discrete Fourier transform of $s(k)$ gives the **Fourier descriptors**

$$a(u) = \sum_{k=0}^{K-1} s(k)\, e^{-j2\pi uk/K}, \qquad u = 0, 1, \dots, K-1,$$

and the inverse rebuilds the boundary. If we keep only P of the coefficients (the lowest frequencies, both positive and negative) and set the rest to zero, we get an approximation $\hat{s}(k)$ that is smooth and keeps the overall shape (Figure 12.2). This is why a handful of Fourier descriptors can represent a shape compactly.

The descriptors react to basic geometric changes in simple ways [1]:

Change to the boundary	Effect on $a(u)$
Translation by $\Delta_{xy} = \Delta_x + j\Delta_y$	only $a(0)$ changes: $a(0) + K\Delta_{xy}$
Rotation by θ	every coefficient is multiplied by $e^{j\theta}$
Scaling by α	every coefficient is multiplied by α
Start point moved by k_0	$a(u)$ is multiplied by $e^{-j2\pi k_0 u/K}$

So we can build an invariant descriptor: drop $a(0)$ (translation), divide by $|a(1)|$ (scale), and keep only magnitudes $|a(u)|$ (rotation and start point). The cost of keeping only magnitudes is losing phase, which carries part of the shape information. This normalization requires a nonzero first harmonic; a symmetric contour or a change of traversal direction can make it vanish. The code rejects a numerically tiny first harmonic rather than dividing by it. Comparing independently traced shapes also requires a common sampling convention and traversal direction; the transformation experiment below keeps the same samples.

```
import numpy as np
from skimage import data, measure

mask = data.horse() == 0
c = max(measure.find_contours(mask.astype(float), 0.5), key=len)
z = c[:, 1] + 1j * c[:, 0]                       # boundary as complex numbers x + jy

def fourier_descriptor(z, n=16):
```

```
    a = np.fft.fft(z)
    a[0] = 0                                    # drop DC   -> translation invariance
    scale = np.abs(a[1])
    if scale <= 1e-12 * np.linalg.norm(a):
        raise ValueError("first harmonic is too small for this normalization")
    a = a / scale                               # divide by |a(1)| -> scale invariance
    return np.abs(np.r_[a[1:n + 1], a[-n:]])    # magnitudes -> rotation and start-point
     ↪  invariance

f1 = fourier_descriptor(z)
f2 = fourier_descriptor(0.5 * np.exp(1j * 0.7) * np.roll(z, 300) + (40 + 10j))
print(np.abs(f1 - f2).max())                    # ~1e-16: same descriptor
```

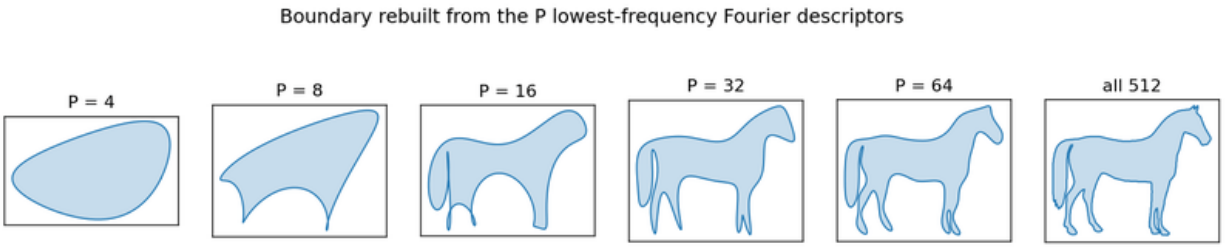


Figure 12.2 — The horse boundary, resampled to 512 points, rebuilt from its P lowest-frequency Fourier descriptors. With $P = 4$ it is an oval blob; by $P = 32$ the legs, neck and head are recognizable; the remaining coefficients mostly add small-scale detail.

Statistical moments of a boundary

In plain words: treat a boundary segment or a signature as a 1-D curve and summarize it with a few statistical moments, such as its spread and its asymmetry [1]. Moments are cheap, physically interpretable, and insensitive to rotation if computed on a rotation-normalized signature.

Online only: Moments of a 1-D boundary function — https://blog.shyandram.dev/en/learn/image-processing/feature-extraction/notes/moments-of-a-1-d-boundary-function/

Region feature descriptors

Plain version. Instead of tracing the edge, we can look at the whole patch: how big it is, how round, how many holes it has, and how its inside looks (smooth, grainy, striped).

Basic descriptors

- **Area** A: the number of pixels in the region.
- **Perimeter** P: the length of its boundary.
- **Compactness**: P^2/A. It is dimensionless, so it is invariant to scale (and to translation and rotation, up to digitization error). A disk minimizes it.

- **Circularity**: $R_c = 4\pi A/P^2$. It equals 1 for a disk and decreases for elongated or ragged regions. It is the inverse of compactness, rescaled.
- **Effective diameter**: $d_e = 2\sqrt{A/\pi}$, the diameter of a disk with the same area.
- **Eccentricity** (from moments): with $\lambda_1 \geq \lambda_2$ the eigenvalues of the covariance matrix of the pixel coordinates, $e = \sqrt{1 - \lambda_2/\lambda_1}$. It is 0 for a disk and approaches 1 for a line segment. (This is the definition used by scikit-image.)

```
import numpy as np
from skimage import data, filters, measure, morphology, segmentation

img = data.coins()
bw = img > filters.threshold_otsu(img)
bw = morphology.binary_closing(bw, morphology.disk(2))
bw = segmentation.clear_border(morphology.remove_small_objects(bw, 200))
for r in measure.regionprops(measure.label(bw))[:4]:
    circ = 4 * np.pi * r.area / r.perimeter ** 2  # 1 for a perfect disk
    print(f"area={r.area:5.0f}  circularity={circ:.2f}  "
          f"eccentricity={r.eccentricity:.2f}  euler={r.euler_number}  "
          f"hu1={r.moments_hu[0]:.4f}")
```

On the coins image, the well-segmented coins have circularity around 0.9 and eccentricity around 0.3–0.4 (digital perimeters are slightly overestimated, so even a perfect digital disk scores a bit below 1).

Topological descriptors

Topology studies properties that survive *rubber-sheet* deformations: stretching and bending without tearing or gluing. Area and perimeter are not topological; the number of pieces and the number of holes are. The **Euler number** is

$$E = C - H,$$

where C is the number of connected components and H the number of holes [1]. The letter "A" has $E = 0$ (one component, one hole); "B" has $E = -1$. Note that E depends on the connectivity convention (4 or 8) used for foreground and background.

Online only: Euler's formula for polygonal networks — https://blog.shyandram.dev/en/learn/image-processing/feature-extraction/notes/eulers-formula-for-polygonal-networks/

Texture

Plain version. Texture is how a surface "feels" to the eye: smooth, coarse, regular. We describe it with statistics of the pixel values and of how pixel values sit next to each other.

Statistical moments of the histogram. Let z be a random variable for intensity, $p(z_i)$ the normalized histogram, $i = 0, \ldots, L-1$, and $m = \sum_i z_i p(z_i)$ the mean. Then [1]:

$$\mu_n(z) = \sum_{i=0}^{L-1} (z_i - m)^n \, p(z_i) \qquad (n\text{-th central moment})$$

$$R(z) = 1 - \frac{1}{1 + \sigma^2(z)} \qquad (\text{relative smoothness; } \sigma^2 = \mu_2)$$

$$U(z) = \sum_{i=0}^{L-1} p^2(z_i) \qquad (\text{uniformity})$$

$$e(z) = -\sum_{i=0}^{L-1} p(z_i) \log_2 p(z_i) \qquad (\text{entropy})$$

R is 0 for a constant region and approaches 1 for high variance (normalize σ^2 to $[0,1]$ first, for example by dividing by $(L-1)^2$). μ_3 measures histogram skew. These measures ignore *where* pixels are, so a checkerboard and its shuffled version get identical values.

Co-occurrence matrices. To capture spatial arrangement, Haralick, Shanmugam and Dinstein proposed the gray-level co-occurrence matrix (GLCM) [3]. Fix a **position operator** Q, for example "one pixel to the right". Then G is an $L \times L$ matrix whose element g_{ij} counts how many times a pixel with level i has a pixel with level j in position Q. Let $n = \sum_{i,j} g_{ij}$ and

$$p_{ij} = g_{ij} / n,$$

the estimated probability of that pair. Useful descriptors computed from p_{ij} include (with m_r, m_c and σ_r, σ_c the means and standard deviations of the row and column marginals):

$$\text{Contrast} = \sum_{i,j} (i-j)^2 \, p_{ij} \qquad \text{Homogeneity} = \sum_{i,j} \frac{p_{ij}}{1 + |i-j|}$$

$$\text{Uniformity (energy)} = \sum_{i,j} p_{ij}^2 \qquad \text{Entropy} = -\sum_{i,j} p_{ij} \log_2 p_{ij}$$

$$\text{Correlation} = \sum_{i,j} \frac{(i - m_r)(j - m_c) \, p_{ij}}{\sigma_r \sigma_c}$$

(scikit-image's `graycoprops` reports "energy" as the square root of the uniformity sum.) Mass near the diagonal means neighbors have similar values (smooth texture, high homogeneity, low contrast). Mass spread away from the diagonal means rapid changes (high contrast). In practice we quantize to few levels (8–32), compute G for several angles and distances, and average over angles for approximate rotation invariance.

```
import numpy as np
from skimage import data
from skimage.feature import graycomatrix, graycoprops

for name in ["brick", "grass", "gravel"]:
    q = (getattr(data, name)()[:256, :256] // 32).astype(np.uint8)   # 8 gray levels
    P = graycomatrix(q, distances=[1], angles=[0, np.pi/4, np.pi/2, 3*np.pi/4],
                     levels=8, symmetric=True, normed=True)
    feats = {p: graycoprops(P, p).mean() for p in ["contrast", "homogeneity", "energy",
     ↪  "correlation"]}
```

```
    print(name, {k: round(float(v), 3) for k, v in feats.items()})
```

```
brick  {'contrast': 0.269, 'homogeneity': 0.883, 'energy': 0.579, 'correlation': 0.817}
grass  {'contrast': 0.995, 'homogeneity': 0.717, 'energy': 0.288, 'correlation': 0.664}
gravel {'contrast': 0.669, 'homogeneity': 0.772, 'energy': 0.323, 'correlation': 0.78}
```

The brick wall is dominated by large flat bricks, so its GLCM is concentrated on a few diagonal cells: high energy, low contrast. Grass changes from blade to blade, so its GLCM spreads out: highest contrast, lowest energy. Four numbers already separate the three textures (Figure 12.3).

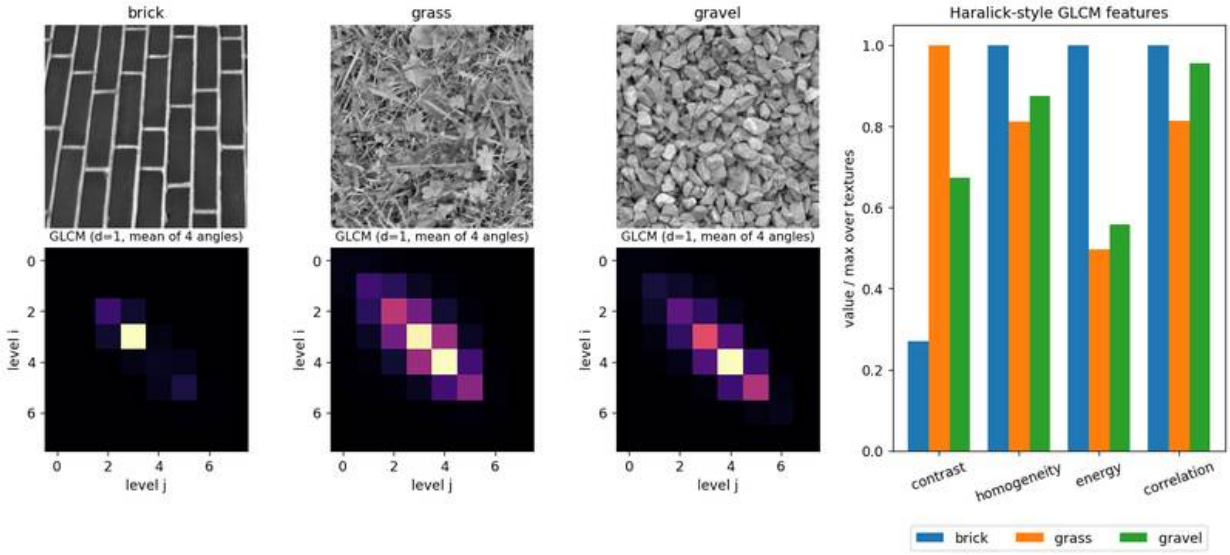


Figure 12.3 — Co-occurrence texture features on three skimage.data textures. Top: 256 × 256 patches. Bottom: GLCMs at distance 1 averaged over four angles (bright = frequent pair). Right: four Haralick-style features, each divided by its maximum over the three textures so they share one axis.

Spectral texture. Periodic or directional texture produces strong, concentrated peaks in the Fourier spectrum. Write the spectrum in polar coordinates, $S(r, \theta)$, where r is the distance from the origin (frequency) and θ the direction. Two 1-D functions summarize it [1]:

$$S(r) = \sum_{\theta=0}^{\pi} S_\theta(r), \qquad S(\theta) = \sum_{r=1}^{R_0} S_r(\theta),$$

where $S_\theta(r)$ is the spectrum along the ray at angle θ, $S_r(\theta)$ the spectrum on the half-circle of radius r, and R_0 the largest radius considered. Peaks in $S(r)$ reveal the period of the pattern (the brick spacing); peaks in $S(\theta)$ reveal its dominant directions. Only half the circle is needed because the spectrum of a real image is symmetric about the origin.

Moment invariants

In plain words: moments summarize how a region's mass is spread around its centre; suitable combinations of them do not change when the region is moved, scaled or rotated.

For a 2-D image (or region indicator) $f(x, y)$ of size $M \times N$, the moment of order $(p + q)$ is

$$m_{pq} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} x^p y^q f(x, y),$$

and the **central moment** is

$$\mu_{pq} = \sum_{x} \sum_{y} (x - \bar{x})^p (y - \bar{y})^q f(x, y), \qquad \bar{x} = \frac{m_{10}}{m_{00}}, \; \bar{y} = \frac{m_{01}}{m_{00}},$$

where $(\bar{x}, \bar{y})$ is the centroid. Central moments are translation invariant. **Normalized central moments**

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{\gamma}}, \qquad \gamma = \frac{p + q}{2} + 1,$$

are also scale invariant. Hu [2] derived seven combinations of the second- and third-order η_{pq} that are invariant to translation, scale *and rotation*. The first two are

$$\phi_1 = \eta_{20} + \eta_{02}, \qquad \phi_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2.$$

ϕ_1 is the normalized "moment of inertia" about the centroid: it grows as mass moves away from the center. ϕ_2 measures elongation. The seventh invariant changes sign under reflection, so it can tell a shape from its mirror image. Because higher-order moments amplify noise, and their values span many orders of magnitude, people usually compare $\operatorname{sign}(\phi_i) \log |\phi_i|$. In scikit-image, `regionprops(...).moments_hu` returns all seven; in OpenCV, `cv2.HuMoments(cv2.moments(mask))`.

Principal components as feature descriptors

Plain version. When we measure many things about an object, some measurements tell the same story twice. PCA finds a new set of axes ordered from "most informative" to "least informative", so we can keep the first few and drop the rest. For a 2-D shape, the first axis is simply the direction in which the shape is longest.

Precise version. Let $\mathbf{x}_1, \ldots, \mathbf{x}_K$ be n-dimensional vectors: for example the n band values of each pixel in a multispectral image, or the (x, y) coordinates of each pixel of a region. Their mean and covariance are

$$\mathbf{m}_x = \frac{1}{K} \sum_{k=1}^{K} \mathbf{x}_k, \qquad \mathbf{C}_x = \frac{1}{K - 1} \sum_{k=1}^{K} (\mathbf{x}_k - \mathbf{m}_x)(\mathbf{x}_k - \mathbf{m}_x)^{\mathsf{T}}.$$

$\mathbf{C}_x$ is real and symmetric, so it has n orthonormal eigenvectors $\mathbf{e}_i$ with eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n \geq 0$. Stack the eigenvectors as rows of a matrix $\mathbf{A}$. The **Hotelling transform** (also called the

discrete Karhunen–Loève transform, or PCA) is

$$\mathbf{y} = \mathbf{A}(\mathbf{x} - \mathbf{m}_x).$$

The new vectors have zero mean and a *diagonal* covariance, $\mathbf{C}_y = \mathbf{A}\mathbf{C}_x\mathbf{A}^{\mathsf{T}} = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$: the components of $\mathbf{y}$ are uncorrelated, and the variance along the i-th axis is λ_i. If we keep only the first k rows, $\mathbf{A}_k$, the reconstruction $\hat{\mathbf{x}} = \mathbf{A}_k^{\mathsf{T}}\mathbf{y} + \mathbf{m}_x$ has mean squared error

$$e_{\text{ms}} = \sum_{j=k+1}^{n} \lambda_j,$$

the sum of the discarded eigenvalues. No other linear projection onto k dimensions does better in this sense.

Three uses in image processing [1]:

1. **Compressing multispectral images.** With n registered bands, each pixel is an n-vector. Most of the variance usually sits in the first two or three principal-component images, so they can replace all n bands for display or classification.
2. **Normalizing object pose.** Use the pixel coordinates of a region as the vectors. Then $\mathbf{y} = \mathbf{A}(\mathbf{x} - \mathbf{m}_x)$ moves the centroid to the origin and rotates the region so that its major axis lies along the first coordinate axis (Figure 12.4). Descriptors computed afterwards are translation and rotation invariant; dividing by $\sqrt{\lambda_1}$ adds scale invariance. One caveat: an eigenvector's sign is arbitrary, so the aligned object may be flipped; a rule such as "the heavier third-moment side points right" resolves it.
3. **Eigen-images.** Flatten each of many same-size images of one object class into a vector. The leading eigenvectors, reshaped back to images, are "eigen-images" that span most of the variation in the set. Any new image can then be described by its handful of coordinates $\mathbf{y}$, a compact feature vector for recognition.

```
import numpy as np
from skimage import data, transform

mask = transform.rotate((data.horse() == 0).astype(float), 35, resize=True) > 0.5
X = np.argwhere(mask)[:, ::-1].astype(float)     # (x, y) of every object pixel
m = X.mean(axis=0)
C = np.cov((X - m).T)                            # 2x2 covariance matrix
evals, evecs = np.linalg.eigh(C)                 # ascending eigenvalues
A = evecs[:, ::-1].T                             # rows = eigenvectors, largest first
Y = (X - m) @ A.T                                # Hotelling transform y = A(x - m)
print(np.round(np.cov(Y.T), 3))                  # diagonal: the axes are decorrelated
```

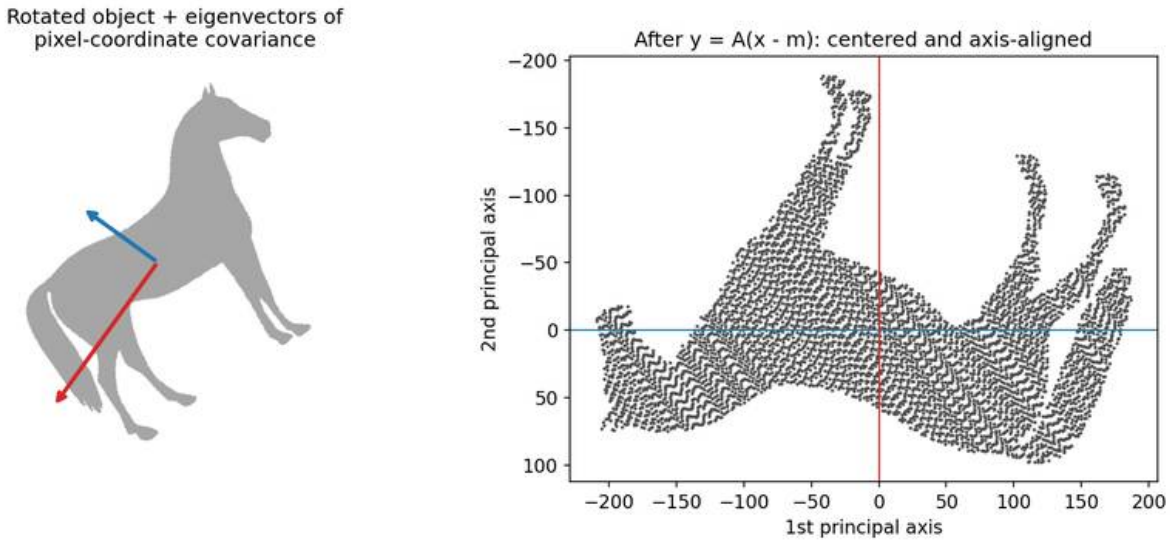


Figure 12.4 — PCA as a pose normalizer. Left: the horse rotated by 35°, with the eigenvectors of its coordinate covariance (arrow length $= 2\sqrt{\lambda}$). Right: the pixel coordinates after $y = A(x - m)$. The object is centered and its longest direction lies on the horizontal axis; it appears flipped because the sign of each eigenvector is arbitrary.

Whole-image features

Plain version. So far we needed a segmented object. Many tasks (stitching panoramas, tracking, 3D reconstruction) instead need *landmarks* that can be found automatically in any photo and found again in another photo of the same scene. Corners and stable blobs are good landmarks.

The Harris–Stephens corner detector

Plain version. Look at the image through a tiny window and slide it a little. In a flat area nothing changes. Along an edge, sliding along the edge changes nothing but sliding across does. At a corner, sliding in *any* direction changes the view. Harris and Stephens turned this into a formula [4].

Precise version. The change in a window w caused by a shift (u, v) is

$$E(u,v) = \sum_{x,y} w(x,y)\,[f(x+u, y+v) - f(x,y)]^2 \approx [u \;\; v]\,\mathbf{M}\begin{bmatrix} u \\ v \end{bmatrix},$$

using a first-order Taylor expansion, with the **structure tensor** (second-moment matrix)

$$\mathbf{M} = \sum_{x,y} w(x,y)\begin{bmatrix} f_x^2 & f_x f_y \\ f_x f_y & f_y^2 \end{bmatrix},$$

where f_x,f_y are the image derivatives and w is a window, usually Gaussian. The eigenvalues λ_1, λ_2 of $\mathbf{M}$ tell the story: both small means flat, one large means edge, both large means corner. Computing eigenvalues at every pixel was costly in 1988, so Harris and Stephens used the **response**

$$R = \det(\mathbf{M}) - k\,\operatorname{tr}(\mathbf{M})^2 = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2,$$

with k a small constant (values around 0.04–0.06 are common). R is large and positive at corners, negative at edges, and near zero in flat areas. Corners are local maxima of R above a threshold, after non-maximum suppression.

Properties: $\mathbf{M}$ is built from derivatives, so R ignores additive brightness changes; its eigenvalues do not depend on rotation, so the detector is rotation covariant. It is **not** scale covariant: a corner seen through a small window may look like an edge through a large one. Scale is what SIFT adds.

```
import numpy as np
from scipy import ndimage as ndi
from skimage import data, feature, util

def harris(img, sigma_d=1.0, sigma_i=1.5, k=0.05):
    Ix = ndi.gaussian_filter(img, sigma_d, order=(0, 1))   # derivative along x (columns)
    Iy = ndi.gaussian_filter(img, sigma_d, order=(1, 0))   # derivative along y (rows)
    Sxx = ndi.gaussian_filter(Ix * Ix, sigma_i)            # entries of the structure tensor
     ↪  M
    Syy = ndi.gaussian_filter(Iy * Iy, sigma_i)
    Sxy = ndi.gaussian_filter(Ix * Iy, sigma_i)
    det, tr = Sxx * Syy - Sxy ** 2, Sxx + Syy
    return det - k * tr ** 2                               # R > 0 corner, R < 0 edge, |R|
     ↪  small flat

img = util.img_as_float(data.camera())
R = harris(img)
corners = feature.corner_peaks(R, min_distance=7, threshold_rel=0.02)   # (row, col) list
```

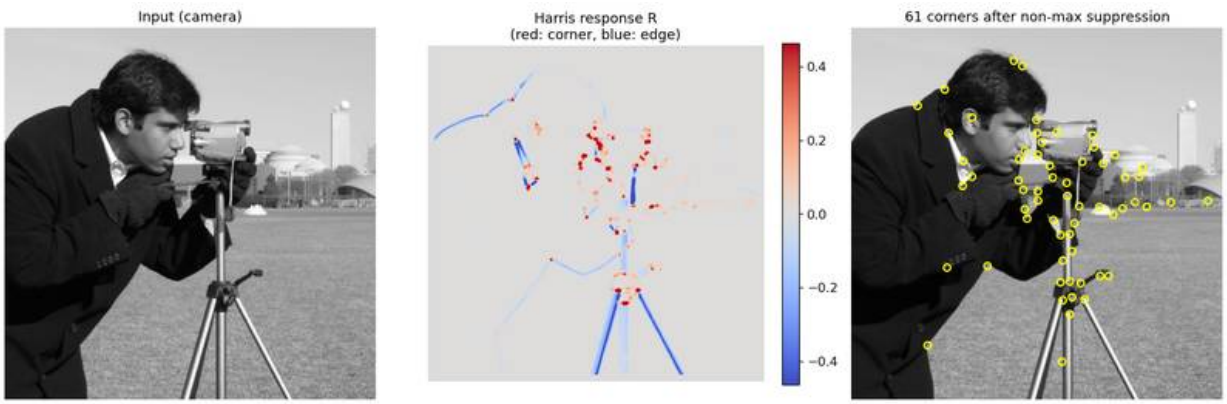


Figure 12.5 — Harris–Stephens corners on skimage.data.camera ($k = 0.05$, integration $\sigma = 1.5$). Middle: the response R; strong positive values (red) cluster on the camera body and tripod joints, while the tripod legs, which are edges, give negative values (blue). Right: local maxima of R after non-maximum suppression.

Interactive demonstration (online): open the online edition of this tutorial to use this demonstration.Open demonstration

Maximally stable extremal regions (MSER)

Plain version. Imagine slowly flooding a gray-level landscape. Dark letters on bright paper become little lakes. As the water level rises, most lakes grow quickly or merge, but a letter's lake keeps exactly its shape over a long range of levels, because the ink is much darker than the paper around it. Regions that barely change while the level changes a lot are *maximally stable*. Matas, Chum, Urban and Pajdla introduced them for wide-baseline stereo [5].

Precise version. Threshold the image at every level t. A connected component of $\{f \le t\}$ (or of $\{f \ge t\}$, for bright regions) is an **extremal region** Q_t: every pixel inside is darker (or brighter) than every pixel on its outer boundary. As t increases, the components grow and merge, forming a tree. For each region, define the stability

$$q(t) = \frac{|Q_{t+\Delta} \setminus Q_{t-\Delta}|}{|Q_t|},$$

where $|\cdot|$ is area and Δ a step in gray levels. A region is **maximally stable** when $q(t)$ has a local minimum. The whole tree can be built in near-linear time using a union-find structure over pixels sorted by intensity.

Why it works: the definition uses only the *order* of intensities, so MSERs are invariant to any monotonic change of brightness. Connected regions map to connected regions under continuous geometric changes, so MSERs are covariant with affine transformations (commonly an ellipse is fitted to each region and normalized to a circle before describing it). MSERs work best on well-defined uniform blobs, such as text, signs and windows, and poorly on blurred or highly textured scenes.

```
import cv2
from skimage import data

img = data.page()                                    # uint8 grayscale
mser = cv2.MSER_create(delta=5, min_area=10, max_area=800, max_variation=0.5)
regions, boxes = mser.detectRegions(img)             # pixel lists + bounding boxes
print(len(regions), "regions")
```

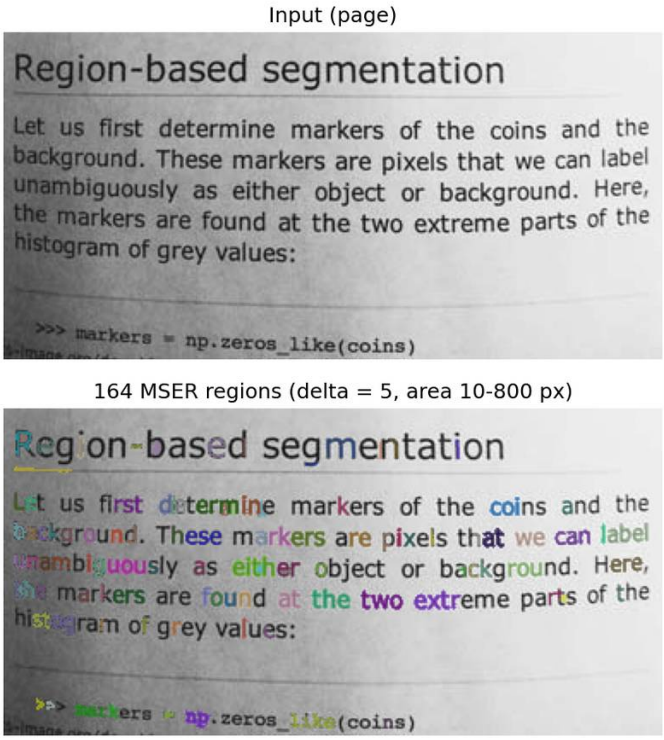


Figure 12.6 — MSER on skimage.data.page, which has uneven lighting. Each colored region is an extremal region that stayed stable over ±5 gray levels. Most are single letters or merged letter groups; because stability depends only on intensity order, the darker bottom-left corner causes no trouble.

Scale-invariant feature transform (SIFT)

Plain version. SIFT, introduced by Lowe [6], answers three questions about every landmark: *where* is it, *how big* is it, and *which way* is it facing? Then it writes down a summary of the gradients around it, measured in that landmark's own size and direction, so the summary is less sensitive to changes of scale, in-plane rotation and contrast. Strong viewpoint or illumination changes can still alter it.

Online only: SIFT: make local features repeatable before trying to match them — https://blog.shyandram.dev/en/learn/image-processing/feature-extraction/papers/sift-make-local-features-repeatable-before-trying-to-match-them/

Scale space

To find features at every size, SIFT builds a **scale space**: the image blurred by Gaussians of increasing width,

$$L(x,y,\sigma) = G(x,y,\sigma) \star f(x,y), \qquad G(x,y,\sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2},$$

where $\star$ denotes convolution and σ the scale. The scales are grouped into **octaves**; each octave doubles σ and is divided into s intervals, so consecutive scales differ by a factor $k = 2^{1/s}$. Lowe [6] uses $s = 3$ and a base scale $\sigma_0 = 1.6$. After each octave the image is downsampled by 2, which keeps the cost low.

Keypoint detection with difference of Gaussians

In plain words: subtracting two blurred copies of the image highlights blobs of one particular size; keypoints are the places where this response is strongest.

Adjacent blurred images are subtracted:

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma).$$

This **difference of Gaussians** (DoG) is a close approximation of the scale-normalized Laplacian of Gaussian, $\sigma^2\nabla^2 G$, scaled by the constant $(k-1)$. It responds strongly to blobs whose size matches σ (Figure 12.7). Candidate keypoints are the pixels whose D value is larger or smaller than all 26 neighbors: 8 in the same DoG image and 9 in each of the scales above and below. A keypoint therefore comes with a position *and* a characteristic scale.

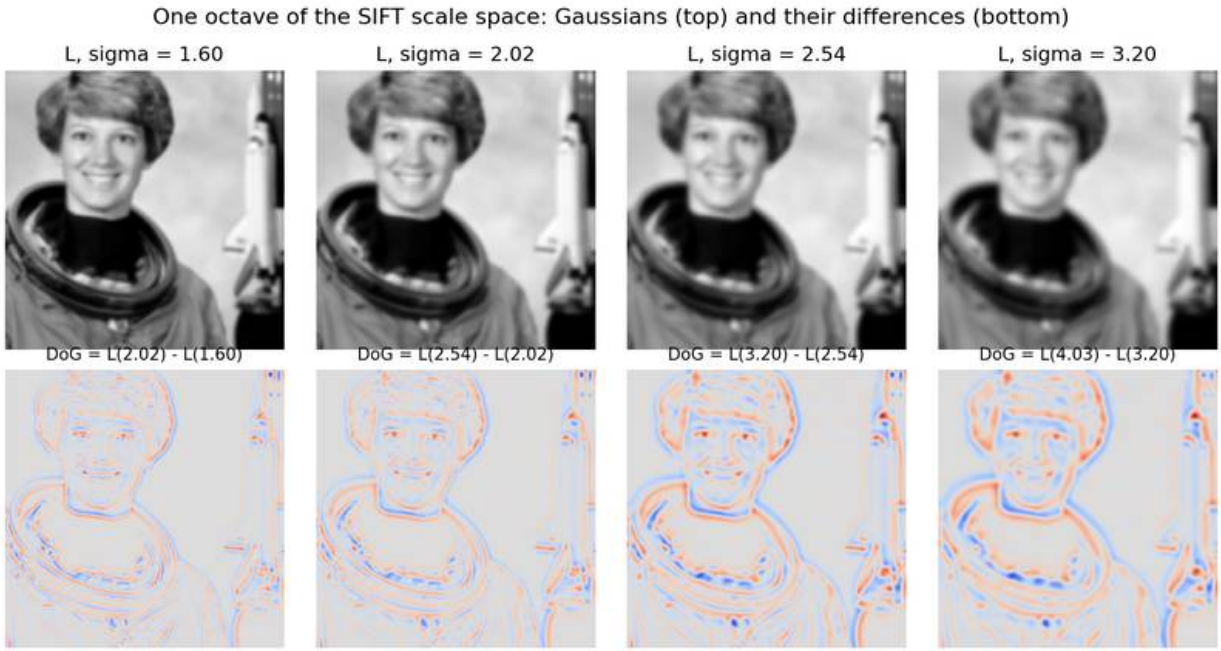


Figure 12.7 — One octave of a SIFT-style scale space on part of skimage.data.astronaut ($\sigma_0 = 1.6$, three intervals per octave). The bottom row shows DoG images: fine structures (hair strands, eyes) respond at small σ, larger structures (the helmet ring, the rocket body) at larger σ. Keypoints are extrema across space and across neighboring DoG images.

Online only: Keypoint localization — https://blog.shyandram.dev/en/learn/image-processing/feature-extraction/notes/keypoint-localization/

Orientation assignment

In plain words: give each keypoint a direction, the dominant gradient direction around it, so that later measurements can be made relative to it.

Around each keypoint, compute gradient magnitudes and directions in L at the keypoint's scale. Build a 36-bin histogram of directions (10° per bin), each sample weighted by its gradient magnitude and by a Gaussian window with σ equal to 1.5 times the keypoint scale. The highest peak gives the keypoint's orientation; any other peak above 80% of the highest creates an additional keypoint with that orientation [6]. From now on, all measurements are made relative to this orientation, which gives rotation invariance.

The keypoint descriptor

In plain words: summarize the gradients around the keypoint in a small grid of direction histograms, measured in the keypoint's own size and direction.

Take a window around the keypoint, rotated to its orientation and sized by its scale. Divide it into a 4×4 grid of cells. In each cell, accumulate an 8-bin histogram of gradient directions, weighted by magnitude and a Gaussian centered on the keypoint, with trilinear interpolation so that small shifts do not cause sudden jumps between bins. This gives $4 \times 4 \times 8 = 128$ numbers [6]. Finally:

1. normalize the vector to unit length (cancels contrast changes $f \mapsto af$; gradients already cancel $+b$),
2. clip every element at 0.2 (limits the influence of a few very large gradients caused by nonlinear lighting such as glare),
3. renormalize to unit length.

Matching

Descriptors from two images are compared by Euclidean distance. The nearest neighbor alone is not reliable: many keypoints have no true match at all. Lowe's **ratio test** keeps a match only if

$$\frac{\|\mathbf{d} - \mathbf{d}_1\|}{\|\mathbf{d} - \mathbf{d}_2\|} < \tau,$$

where $\mathbf{d}_1, \mathbf{d}_2$ are the nearest and second-nearest descriptors in the other image and τ a threshold (Lowe suggests 0.8) [6]. A distinctive match is much closer than the runner-up. Large databases use approximate nearest-neighbor search, and the surviving matches are usually verified geometrically, for example by fitting a similarity, affine, or projective transformation with RANSAC (random sample consensus) [21] and keeping the inliers. RANSAC is introduced in the Supplementary notes (https://blog.shyandram.dev/en/learn/image-processing/feature-extraction/notes/ransac-fitting-a-model-despite-outliers/).

```
import numpy as np
from skimage import data, feature, transform
```

```
from skimage.color import rgb2gray
from skimage.measure import ransac

img1 = rgb2gray(data.astronaut())
tf = transform.AffineTransform(scale=0.7, rotation=np.deg2rad(25), translation=(140, -60))
img2 = 0.8 * transform.warp(img1, tf.inverse) + 0.1        # second "view"

sift = feature.SIFT()
sift.detect_and_extract(img1); k1, d1 = sift.keypoints, sift.descriptors
sift.detect_and_extract(img2); k2, d2 = sift.keypoints, sift.descriptors

matches = feature.match_descriptors(d1, d2, max_ratio=0.6, cross_check=True)  # ratio test
src, dst = k1[matches[:, 0]][:, ::-1], k2[matches[:, 1]][:, ::-1]           # (row, col) ->
 ↪  (x, y)
model, inliers = ransac((src, dst), transform.SimilarityTransform,
                        min_samples=3, residual_threshold=2, max_trials=500)
print(len(matches), "matches,", inliers.sum(), "inliers")
print("recovered scale %.3f, rotation %.1f deg" % (model.scale, np.degrees(model.rotation)))
```

On this synthetic pair the script finds 484 matches, 483 of them consistent with the true transformation, and recovers scale 0.700 and rotation 25.0°. Real photo pairs, with viewpoint changes, occlusion and repeated patterns, give far lower inlier ratios, which is why geometric verification is not optional.

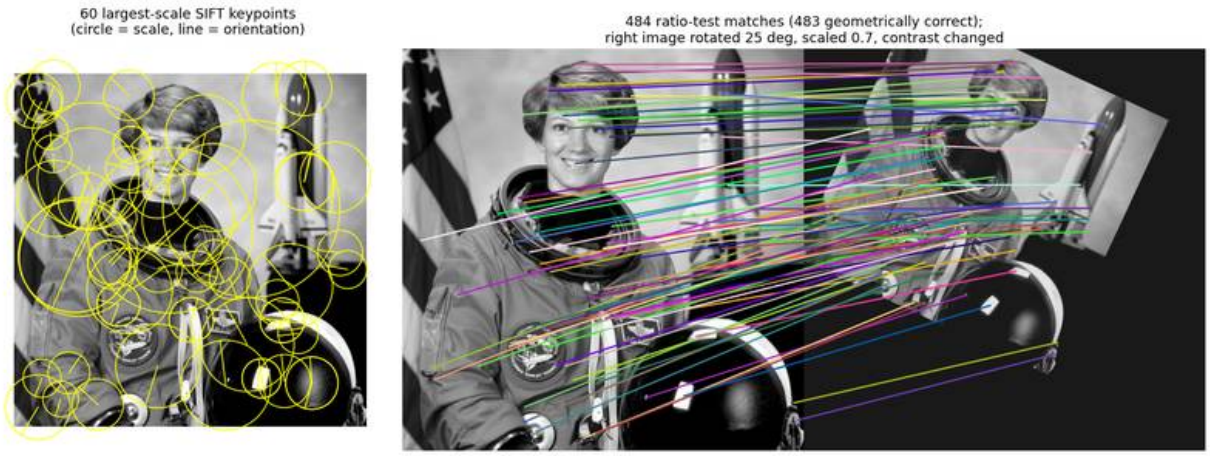


Figure 12.8 — SIFT on skimage.data.astronaut. Left: the 60 largest-scale keypoints (circle radius ∝ scale, line = orientation). Right: a subset of ratio-test matches to a copy rotated by 25°, scaled by 0.7 and changed in contrast; 483 of the 484 matches agree with the true transformation within 3 pixels.

Where to find SIFT

scikit-image added a SIFT detector and descriptor in version 0.19 (`skimage.feature.SIFT`, used above). In OpenCV, call `cv2.SIFT_create()` and then `detectAndCompute(gray, None)`.

Modern view

What the surveys say

Shape descriptors. Zhang and Lu [9] sort shape descriptors into contour-based versus region-based and global versus structural; every boundary and region descriptor of this chapter fits that grid, and no single descriptor is best for every application.

Local feature detectors. Tuytelaars and Mikolajczyk [10] organize detectors around the properties a good local feature should have, above all repeatability, which can be reached by invariance or by robustness; some properties compete, so the right compromise depends on the application. The speed-oriented descendants of SIFT illustrate the trade-off: SURF [7] uses box filters on integral images, and ORB [8] pairs the FAST corner test with a binary descriptor.

Texture. Liu et al. [11] review about two decades of texture representation, from bag-of-words pipelines built on hand-designed local descriptors to CNN-based representations; co-occurrence statistics [3] sit at the root of this history.

Image matching, handcrafted to deep. Ma et al. [12] show that learning has entered every stage of the matching pipeline. Jin et al. [13] score methods by the accuracy of the final camera poses and find that properly tuned classical pipelines such as SIFT can still beat methods perceived as the state of the art.

Online only: Guide to the Modern view surveys — https://blog.shyandram.dev/en/learn/image-processing/feature-extraction/papers/guide-to-the-modern-view-surveys/

How deep learning changed feature extraction

Learned keypoints and descriptors. SuperPoint [14] trains one network to output both a keypoint heatmap and dense descriptors, self-supervised through *homographic adaptation*; DISK [16] trains detection and description end to end with reinforcement learning. They replace the formulas of Harris and SIFT with learned functions but keep the same idea: sparse, repeatable points with descriptors.

Learned matching. SuperGlue [15] and its faster successor LightGlue [18] let the keypoints of both images attend to each other instead of applying the ratio test, and solve an assignment that can also say "no match". LoFTR [17] removes the detector and matches dense feature maps with a Transformer, which helps in low-texture areas.

General-purpose deep features. Self-supervised foundation models such as DINOv2 [19] produce patch features that work across many tasks without fine-tuning, and matchers such as RoMa [20] build on them: the handcrafted descriptor at the center of SIFT has been replaced by a network trained on a very large image collection.

Online only: Learned features in more detail — https://blog.shyandram.dev/en/learn/image-processing/feature-extraction/notes/learned-features-in-more-detail/

What did not change. The *problem decomposition* of this chapter survives. Modern pipelines still

detect, describe, match and verify geometrically, typically still with a RANSAC-style robust estimator [21]. Scale and rotation still have to be handled, either built in (SIFT's scale space, ORB's orientation) or learned from data augmentation. Invariance is still a design choice: a descriptor that is too invariant loses discriminative power, whether it was handcrafted or learned. And in measurement-driven fields (materials, microscopy, quality inspection), region descriptors such as area, circularity, Euler number and GLCM statistics remain popular because each number has a physical meaning a human can check.

Key takeaways

- A **detector** finds features (points, regions, boundaries); a **descriptor** turns each into a vector. Detectors should be covariant with geometric changes; descriptors should be invariant to the changes that do not matter for the task.
- Boundaries are first ordered (boundary following) and simplified (chain codes, minimum-perimeter polygons, merging and splitting, signatures, skeletons) before being measured.
- **Fourier descriptors** compress a closed contour into a few coefficients; dropping $a(0)$, dividing by $|a(1)|$ and keeping magnitudes gives translation, scale, rotation and start-point invariance.
- Region descriptors range from size and shape (area, circularity, eccentricity) to topology (Euler number $E = C - H$), texture (histogram moments, **GLCM** features, spectral signatures) and **Hu's moment invariants**.
- **PCA** (the Hotelling transform) decorrelates measurements, aligns objects to their principal axes, and gives compact eigen-image descriptors; the error from dropping components equals the sum of the dropped eigenvalues.
- **Harris** finds corners from the structure tensor; **MSER** finds regions stable over many thresholds; **SIFT** adds scale space, orientation and a normalized 128-D gradient histogram, then matches with the ratio test.
- Learned detectors, matchers and foundation-model features are now the focus of most matching research, but carefully tuned classical pipelines remain strong baselines, and the detect–describe–match–verify structure has not changed.

Exercises

1. **Chain codes by hand.** Write the 8-direction chain code of an axis-aligned 3×2 rectangle of boundary links (3 links wide, 2 tall), starting at its lower-left corner and walking counterclockwise. Compute its first difference and the shape number (minimum circular rotation). Then rotate the rectangle by 90° and show the shape number does not change.

   Hint

   Walking counterclockwise from the lower-left corner (with 0 = east, 2 = north): 0 0 0 2 2 4 4 4 6 6. Each element of the first difference counts counterclockwise turns from the previous direction: 2 0 0 2 0 2 0 0 2 0 (the leading 2 comes from the wrap-around, $6 \to 0$). Its smallest circular rotation is 0 0 2 0 2 0 0 2 0 2. After a 90° turn every code element increases by 2 (mod 8), which leaves the differences unchanged.

2. **Fourier descriptor invariance, tested.** Take the `fourier_descriptor` function from this chapter. (a) Show numerically that it fails to distinguish a shape from its mirror image. (b) Explain why from the table of transformation effects, and propose a change that would tell mirror images apart.

   Hint

   Flip the mask with `mask[:, ::-1]` and trace both contours with `find_contours`; the descriptors agree to within about 10^{-4}. A tracer walks every boundary in the same rotational sense, so the mirrored boundary is $s'(k) = \overline{s(-k)}$ (conjugate *and* reversed), up to a shift. Its DFT is $\overline{a(u)}$: the same magnitudes, opposite phases. Magnitudes cannot see the reflection. Keep some phase instead: the complex number $a(u)\,a(2-u)/a(1)^2$ (indices modulo K) is unchanged by translation, rotation, scale and start point, but it is conjugated by a reflection, so the sign of its imaginary part tells a shape from its mirror image.

3. **Circularity on a grid.** Compute $4\pi A/P^2$ for digital disks of radius 5, 10, 20 and 50 pixels using `skimage.draw.disk` and `regionprops`. Why is the result not exactly 1, and how does it change with radius? Try `perimeter_crofton` as well.

   Hint

   The digital perimeter is a staircase, and different estimators correct for this differently. Errors are relatively larger for small disks. Report both perimeter estimates; one of them should be noticeably closer to the true $2\pi r$.

4. **Texture direction.** Create a synthetic texture of horizontal stripes with period 8 pixels plus a little noise. Compute GLCM contrast for offsets of 1 pixel at angles 0°, 45°, 90° and 135°, and the angular spectral signature $S(\theta)$. Which angle stands out in each, and why do the two methods agree?

   Hint

   Moving along a stripe (0°) keeps intensity almost constant, so contrast is low; moving across stripes (90°) changes it. In the spectrum, horizontal stripes put energy on the vertical frequency axis. Watch the convention: scikit-image's angle 0 means a horizontal pixel offset.

5. **PCA by eigenvalues.** For a filled ellipse with semi-axes $a = 40$ and $b = 10$, predict the ratio λ_1/λ_2 of the coordinate-covariance eigenvalues, then verify it in code. Use your answer to relate λ_1, λ_2 to the eccentricity reported by `regionprops`.

   Hint

   For a uniform filled ellipse, the variance along a semi-axis of length a is $a^2/4$. So $\lambda_1/\lambda_2 = (a/b)^2 = 16$, and $e = \sqrt{1 - \lambda_2/\lambda_1} = \sqrt{1 - b^2/a^2} \approx 0.968$.

6. **Ratio test trade-off.** Using the SIFT matching script, vary `max_ratio` from 0.5 to 0.95 and plot (a) the number of matches and (b) the fraction of RANSAC inliers. Then add a 0.6× rescale plus 30° rotation plus Gaussian noise ($\sigma = 0.05$) and repeat. Where would you set the threshold, and why does the answer depend on the noise?

Hint

A loose ratio admits more matches but more of them are wrong; a strict ratio keeps few, mostly correct ones. RANSAC can tolerate many outliers but needs a minimum number of correct matches, so the best threshold sits between the two extremes and moves with image quality.

References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018, Ch. 12. publisher page
2. M.-K. Hu, "Visual pattern recognition by moment invariants," *IRE Transactions on Information Theory*, vol. 8, no. 2, pp. 179–187, 1962. doi
3. R. M. Haralick, K. Shanmugam, and I. Dinstein, "Textural features for image classification," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-3, no. 6, pp. 610–621, 1973. doi
4. C. Harris and M. Stephens, "A combined corner and edge detector," in *Proc. Alvey Vision Conference*, pp. 23.1–23.6, 1988. doi
5. J. Matas, O. Chum, M. Urban, and T. Pajdla, "Robust wide-baseline stereo from maximally stable extremal regions," *Image and Vision Computing*, vol. 22, no. 10, pp. 761–767, 2004. doi
6. D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004. doi
7. H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (SURF)," *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346–359, 2008. doi
8. E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," in *Proc. IEEE International Conference on Computer Vision (ICCV)*, pp. 2564–2571, 2011. doi
9. D. Zhang and G. Lu, "Review of shape representation and description techniques," *Pattern Recognition*, vol. 37, no. 1, pp. 1–19, 2004. doi
10. T. Tuytelaars and K. Mikolajczyk, "Local invariant feature detectors: A survey," *Foundations and Trends in Computer Graphics and Vision*, vol. 3, no. 3, pp. 177–280, 2008. doi
11. L. Liu, J. Chen, P. Fieguth, G. Zhao, R. Chellappa, and M. Pietikäinen, "From BoW to CNN: Two decades of texture representation for texture classification," *International Journal of Computer Vision*, vol. 127, pp. 74–109, 2019. arXiv
12. J. Ma, X. Jiang, A. Fan, J. Jiang, and J. Yan, "Image matching from handcrafted to deep features: A survey," *International Journal of Computer Vision*, vol. 129, pp. 23–79, 2021. doi
13. Y. Jin, D. Mishkin, A. Mishchuk, J. Matas, P. Fua, K. M. Yi, and E. Trulls, "Image matching across wide baselines: From paper to practice," *International Journal of Computer Vision*, 2021. doi ·arXiv
14. D. DeTone, T. Malisiewicz, and A. Rabinovich, "SuperPoint: Self-supervised interest point detection and description," arXiv:1712.07629, 2017. arXiv

15. P.-E. Sarlin, D. DeTone, T. Malisiewicz, and A. Rabinovich, “SuperGlue: Learning feature matching with graph neural networks,” arXiv:1911.11763, 2019. arXiv
16. M. J. Tyszkiewicz, P. Fua, and E. Trulls, “DISK: Learning local features with policy gradient,” arXiv:2006.13566, 2020. arXiv
17. J. Sun, Z. Shen, Y. Wang, H. Bao, and X. Zhou, “LoFTR: Detector-free local feature matching with Transformers,” arXiv:2104.00680, 2021. arXiv
18. P. Lindenberger, P.-E. Sarlin, and M. Pollefeys, “LightGlue: Local feature matching at light speed,” arXiv:2306.13643, 2023. arXiv
19. M. Oquab, T. Darcet, T. Moutakanni, et al., “DINOv2: Learning robust visual features without supervision,” arXiv:2304.07193, 2023. arXiv
20. J. Edstedt, Q. Sun, G. Bökman, M. Wadenbäck, and M. Felsberg, “RoMa: Robust dense feature matching,” in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024. arXiv
21. M. A. Fischler and R. C. Bolles, “Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography,” *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981. doi
22. scikit-image contributors, “skimage.measure.ransac,” *scikit-image 0.26 documentation (API reference)*. docs

IP 13 • Image Pattern Classification

From minimum-distance and Bayes classifiers to perceptrons, backpropagation and convolutional neural networks: how a computer decides what an image or region shows.

Online edition: https://blog.shyandram.dev/en/learn/image-processing/pattern-classification/

What you'll learn

- How patterns, pattern vectors and pattern classes turn "what is in this image?" into a precise decision problem
- How prototype matching works: the minimum-distance classifier, correlation-based template matching, SIFT feature matching and matching of structural descriptions
- Why the Bayes classifier is optimal, and what it becomes when each class is Gaussian
- How a perceptron learns, how multilayer networks are trained with backpropagation, and how a convolutional neural network (CNN) learns its own features
- The practical side of training: overfitting, data augmentation and regularization
- How image classification moved from AlexNet to Vision Transformers, CLIP and DINOv2, and why benchmark numbers deserve suspicion

The big picture

> **Explain like I'm five**
>
> Imagine a pile of fruit on the kitchen table and three bowls labelled *apples*, *limes* and *oranges*. You pick up each fruit, look at its color and feel its size, and drop it in the bowl it looks most like. That is pattern classification. The color and the size are the **features**. The bowls are the **classes**. The rule in your head ("red and big means apple") is the **classifier**. This chapter is about how to build that rule: by comparing with a typical example of each fruit, by using probabilities, or by letting a network of tiny adding machines learn the rule from many examples.

IP 10–12 carved an image into regions and described each region with numbers: areas, moments, Fourier descriptors, texture statistics, keypoint descriptors. This chapter closes the pipeline. It takes those descriptions and assigns each one a **label**: "coin", "digit 7", "water", "tumor", "cat". Almost every practical vision system ends with such a decision, so this is where image processing meets machine learning.

The chapter follows a historical arc that is also a conceptual one [1]. First come classifiers that work directly with **prototypes** (a typical member of each class). Then comes the **statistically optimal** classifier, which needs the class probability distributions. Finally come **neural networks**, which learn the decision rule from data. In their convolutional form they also learn the features, which removes much of the hand-crafted feature engineering of IP 12. If you want a gentle refresher on

supervised learning, cost functions and gradient descent first, see ML 01 ·Introduction to Machine Learning.

Background

Plain version. A classifier is a function that looks at a description of something and answers "which group does this belong to?". We build it from examples whose answers we already know.

A **pattern** is an arrangement of descriptors that represents an object or region. A **pattern class** is a family of patterns that share some property. We write the W classes as $\omega_1, \omega_2, \dots, \omega_W$. **Pattern recognition** by machine means assigning each pattern to its class automatically and with as few mistakes as possible.

The work splits into two stages:

1. **Representation.** Choose what to measure: pixel values, region descriptors, keypoint descriptors, or learned features. IP 12 covered this stage.
2. **Decision.** Given the measurements, choose a class. This chapter covers this stage.

Methods are usually trained with **supervised learning**: we have a **training set** of patterns with known labels and use it to set the classifier's parameters. We then measure performance on a separate **test set** that the classifier never saw. A third split, the **validation set**, is used to choose settings such as the network size or the stopping time. Mixing these sets up is the most common source of over-optimistic results.

A useful way to think about any classifier is through **decision functions**. For each class ω_j we define a function $d_j(\mathbf{x})$ and assign $\mathbf{x}$ to the class with the largest value:

$$\mathbf{x} \in \omega_i \quad \text{if} \quad d_i(\mathbf{x}) > d_j(\mathbf{x}) \quad \text{for all } j \neq i.$$

Here $\mathbf{x}$ is the pattern and d_j is a scalar score for class ω_j. The **decision boundary** between classes i and j is the set where $d_i(\mathbf{x}) - d_j(\mathbf{x}) = 0$. Every classifier in this chapter, from the simplest to a deep network, is a particular way of building these functions.

Patterns and pattern classes

Plain version. We describe objects either with a list of numbers (a vector) or with a description of how their parts connect (a string or a tree).

Pattern vectors

The most common representation is a column vector of n measurements:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

where each x_k is one descriptor, such as the area of a region, its hue, a moment invariant or the response of a texture filter. A vector is a point in n-dimensional **feature space**. Good features put patterns from the same class close together and patterns from different classes far apart. If the features are poor, no classifier can rescue them.

Two practical points matter here. First, features with different units need **normalization**; otherwise the one with the largest numeric range dominates any distance. In the fruit example, hue lies in $[0, 1]$ while diameter is measured in centimetres, so we standardize each feature to zero mean and unit variance before comparing distances. Second, the whole image can itself be a pattern vector: an $M \times N$ image flattened into an MN-dimensional vector. This is how neural networks see images, and it is why they need so much data, because the space is enormous.

> **Interactive demonstration (online)**: open the online edition of this tutorial to use this demonstration.Open demonstration

Online only: Structural patterns: strings and trees — https://blog.shyandram.dev/en/learn/image-processing/pattern-classification/notes/structural-patterns-strings-and-trees/

Pattern classification by prototype matching

Plain version. Keep one typical example of each class. To classify something new, see which typical example it looks most like.

Minimum-distance classifier

The simplest prototype is the **mean vector** of each class, computed from its training patterns:

$$\mathbf{m}_j = \frac{1}{n_j} \sum_{\mathbf{x} \in \omega_j} \mathbf{x}, \qquad j = 1, 2, \ldots, W,$$

where n_j is the number of training patterns of class ω_j. A new pattern goes to the class whose mean is nearest in Euclidean distance, $D_j(\mathbf{x}) = \|\mathbf{x} - \mathbf{m}_j\|$.

Minimizing $D_j^2 = \mathbf{x}^\top \mathbf{x} - 2\mathbf{x}^\top \mathbf{m}_j + \mathbf{m}_j^\top \mathbf{m}_j$ over j is the same as maximizing

$$d_j(\mathbf{x}) = \mathbf{x}^\top \mathbf{m}_j - \tfrac{1}{2}\, \mathbf{m}_j^\top \mathbf{m}_j,$$

because $\mathbf{x}^\top \mathbf{x}$ does not depend on j. This d_j is linear in $\mathbf{x}$. The boundary between classes i and j is

$$d_i(\mathbf{x}) - d_j(\mathbf{x}) = \mathbf{x}^\top (\mathbf{m}_i - \mathbf{m}_j) - \tfrac{1}{2} (\mathbf{m}_i - \mathbf{m}_j)^\top (\mathbf{m}_i + \mathbf{m}_j) = 0,$$

which is the **perpendicular bisector** of the segment joining $\mathbf{m}_i$ and $\mathbf{m}_j$: a line in 2-D, a plane in 3-D, a hyperplane in general. Figure 13.1 shows the result on our fruit data.

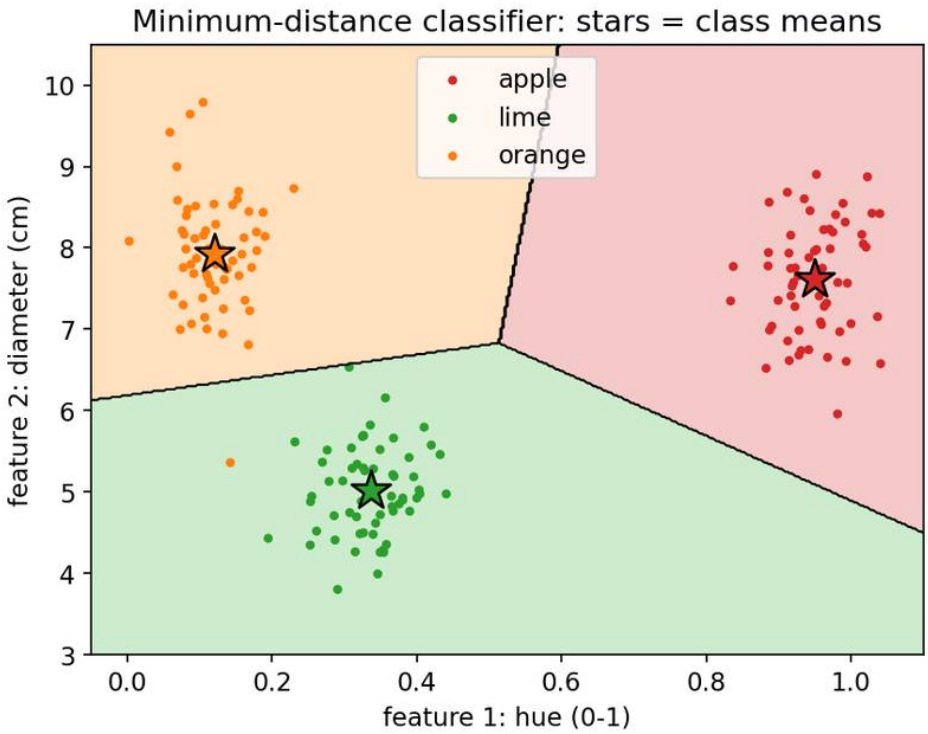


Figure 13.1 — Minimum-distance classification of synthetic fruit (hue, diameter). Stars are the class means; the boundaries are perpendicular bisectors between means (computed on standardized features).

```
import numpy as np

def fit_min_distance(X, y):
    """X: (n, d) pattern vectors, y: (n,) labels -> class list and mean vectors."""
    classes = np.unique(y)
    M = np.stack([X[y == c].mean(axis=0) for c in classes])
    return classes, M

def predict_min_distance(X, classes, M):
    d2 = ((X[:, None, :] - M[None, :, :]) ** 2).sum(axis=-1)  # (n, W) squared distances
    return classes[d2.argmin(axis=1)]

rng = np.random.default_rng(0)
X = np.r_[rng.normal([0, 0], 1, (100, 2)), rng.normal([4, 3], 1, (100, 2))]
y = np.repeat([0, 1], 100)
classes, M = fit_min_distance(X, y)
print("training accuracy:", (predict_min_distance(X, classes, M) == y).mean())
```

The classifier works well when classes are compact, roughly round blobs whose means are far apart compared with their spread. It fails when a class is elongated, curved or made of several

clusters, because one mean cannot represent it. Two simple extensions help: keep **several prototypes per class**, or keep **all** training patterns and use the nearest one (the nearest-neighbor rule).

Matching by correlation (template matching)

When the prototype is a small image rather than a vector, we compare it with every position of a larger image. This is the 2-D version of prototype matching. Raw correlation $\sum wf$ is biased toward bright regions, so we use the **normalized correlation coefficient**:

$$\gamma(x,y) = \frac{\sum_{s,t}\left[w(s,t)-\bar{w}\right]\left[f(x+s,y+t)-\bar{f}_{xy}\right]}{\sqrt{\sum_{s,t}\left[w(s,t)-\bar{w}\right]^2 \sum_{s,t}\left[f(x+s,y+t)-\bar{f}_{xy}\right]^2}},$$

where w is the template, $\bar{w}$ its mean, f the image, and $\bar{f}_{xy}$ the mean of the image patch under the template when its corner is at (x,y). The sums run over the template's support. Because of the subtraction and the division, γ lies in $[-1,1]$ and does not change if the patch's brightness is scaled or offset. A value of 1 means a perfect match up to brightness and contrast. Note that this operation is correlation, not convolution: the template is not flipped (see ML 03 ·Convolution, Image Filtering and CNNs).

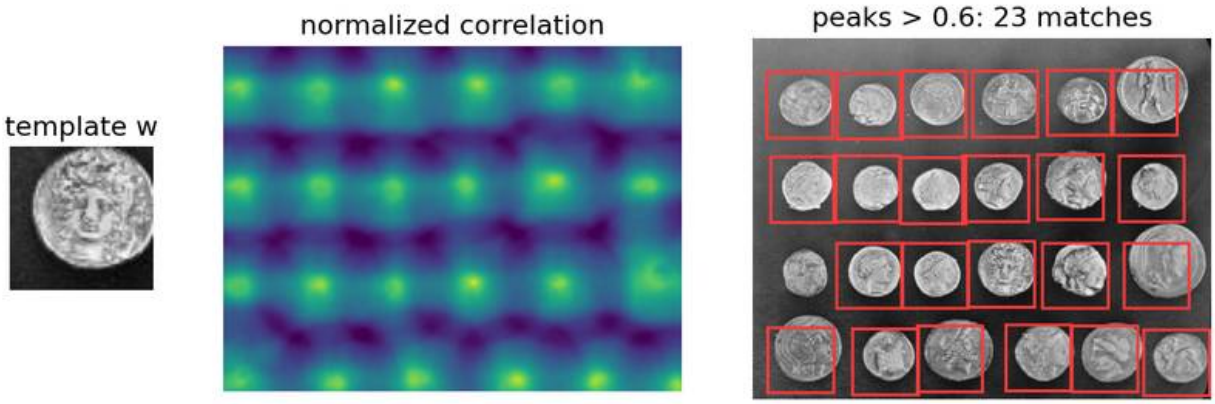


Figure 13.2 — Template matching on skimage's coins image. Left: one coin used as template. Middle: the correlation map, bright where the image looks like the template. Right: local maxima above 0.6.

```
import cv2
import numpy as np
from skimage import data

img = data.coins().astype(np.float32)
w = img[170:225, 180:235]                          # one coin as the template
R = cv2.matchTemplate(img, w, cv2.TM_CCOEFF_NORMED)  # values in [-1, 1]
y0, x0 = np.unravel_index(R.argmax(), R.shape)
print("best match at", (int(x0), int(y0)), "score", round(float(R.max()), 3))
```

Figure 13.2 also shows the method's limits. The template finds almost every coin, because every coin is a bright disk of similar size on a dark background. It cannot tell one coin design from another at this resolution. Correlation is also sensitive to rotation and scale: a template rotated by 30° or enlarged by 30% may no longer produce a clear peak. The usual remedies are to search over a set of rotated and scaled templates, or to match invariant features instead of raw pixels.

Matching SIFT features

The scale-invariant feature transform (SIFT) from IP 12 [3] gives exactly those invariant features. Each keypoint has a location, scale, orientation and a 128-dimensional descriptor. To decide whether a prototype object appears in a scene image:

1. Extract keypoints and descriptors from the prototype and from the scene.
2. For each prototype descriptor, find its nearest and second-nearest scene descriptors in Euclidean distance.
3. Accept the match only if the nearest distance is clearly smaller than the second-nearest, for example $d_1/d_2 < 0.8$. This **ratio test** [3] rejects ambiguous matches in repetitive texture.
4. Check that the accepted matches agree on one geometric transformation (a similarity, affine or projective map). Matches that disagree are outliers.

The decision is then "object present" if enough geometrically consistent matches survive. This is still prototype matching: the prototype is the set of descriptors, and the "distance" is the number of consistent correspondences. It tolerates rotation, scale change, partial occlusion and moderate viewpoint change, which template matching cannot.

Online only: Matching structural prototypes — https://blog.shyandram.dev/en/learn/image-processing/pattern-classification/notes/matching-structural-prototypes/

Optimum (Bayes) statistical classifiers

Plain version. If we know how often each class occurs and what its measurements usually look like, there is a best possible rule: pick the class that is most probable given what we measured. No other rule makes fewer mistakes on average.

Derivation

In plain words: give every possible mistake a cost, and for each pattern choose the class whose expected cost is smallest.

Let $p(\omega_k \mid \mathbf{x})$ be the probability that pattern $\mathbf{x}$ comes from class ω_k. Let L_{kj} be the **loss** incurred when a pattern that truly belongs to ω_k is assigned to ω_j. The expected loss of choosing class ω_j, called the **conditional risk**, is

$$r_j(\mathbf{x}) = \sum_{k=1}^{W} L_{kj}\, p(\omega_k \mid \mathbf{x}).$$

Bayes' rule, $p(\omega_k \mid \mathbf{x}) = p(\mathbf{x} \mid \omega_k)\, P(\omega_k)/p(\mathbf{x})$, rewrites this in terms of the **class-conditional density** $p(\mathbf{x} \mid \omega_k)$ (what patterns of class ω_k look like) and the **prior** $P(\omega_k)$ (how common the class is). The factor $1/p(\mathbf{x})$ is the same for every j, so it can be dropped when comparing risks.

The classifier that picks the j with the smallest $r_j(\mathbf{x})$ for every $\mathbf{x}$ minimizes the total average loss. That is why it is called **optimal**: it is the best any rule can do, given the true distributions. With the common **0–1 loss** ($L_{kj} = 0$ if $k = j$, otherwise 1), every error costs the same and

$$r_j(\mathbf{x}) \propto p(\mathbf{x}) - p(\mathbf{x} \mid \omega_j) P(\omega_j).$$

Minimizing it means maximizing the decision function

$$d_j(\mathbf{x}) = p(\mathbf{x} \mid \omega_j)\, P(\omega_j).$$

In words: choose the class with the largest posterior probability. The catch is that we never know the true densities. We must estimate them from training data, and the classifier is only as good as those estimates.

Bayes classifier for Gaussian pattern classes

In plain words: if the measurements of each class form a bell-shaped cloud, the optimal rule becomes an explicit formula in each cloud's centre and spread.

Assume each class is a multivariate Gaussian with mean $\mathbf{m}_j$ and covariance matrix $\mathbf{C}_j$:

$$p(\mathbf{x} \mid \omega_j) = \frac{1}{(2\pi)^{n/2} |\mathbf{C}_j|^{1/2}} \exp\Big[-\tfrac{1}{2} (\mathbf{x} - \mathbf{m}_j)^{\top} \mathbf{C}_j^{-1} (\mathbf{x} - \mathbf{m}_j) \Big],$$

where n is the dimension of $\mathbf{x}$ and $|\mathbf{C}_j|$ the determinant of $\mathbf{C}_j$. Because the logarithm is increasing, we can maximize $\ln d_j$ instead and drop the constant $\frac{n}{2} \ln 2\pi$:

$$d_j(\mathbf{x}) = \ln P(\omega_j) - \tfrac{1}{2} \ln |\mathbf{C}_j| - \tfrac{1}{2} (\mathbf{x} - \mathbf{m}_j)^{\top} \mathbf{C}_j^{-1} (\mathbf{x} - \mathbf{m}_j).$$

The last term is half the squared **Mahalanobis distance**, a distance that stretches space so that each class's spread looks round. This d_j is quadratic in $\mathbf{x}$, so boundaries between classes are quadrics: ellipses, parabolas or hyperbolas in 2-D. Two special cases connect back to earlier sections:

- **Equal covariances** ($\mathbf{C}_j = \mathbf{C}$ for all j). The quadratic term $\mathbf{x}^{\top} \mathbf{C}^{-1} \mathbf{x}$ is common to all classes and cancels, leaving a linear function
  $d_j(\mathbf{x}) = \ln P(\omega_j) + \mathbf{x}^{\top} \mathbf{C}^{-1} \mathbf{m}_j - \frac{1}{2} \mathbf{m}_j^{\top} \mathbf{C}^{-1} \mathbf{m}_j$. Boundaries become hyperplanes.
- **Identity covariance and equal priors** ($\mathbf{C} = \mathbf{I}$, $P(\omega_j) = 1/W$). The function reduces to $\mathbf{x}^{\top} \mathbf{m}_j - \frac{1}{2} \mathbf{m}_j^{\top} \mathbf{m}_j$: the minimum-distance classifier. So minimum distance is the Bayes-optimal rule for round, equally spread, equally likely Gaussian classes, and only then.

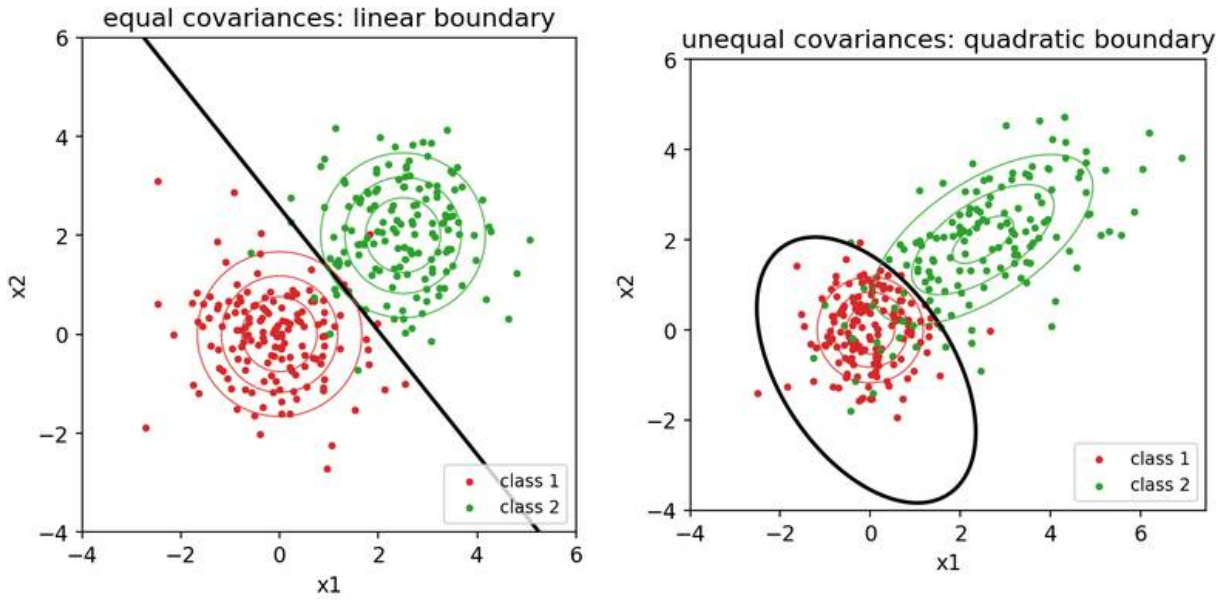


Figure 13.3 — Bayes decision boundaries (thick black) for two Gaussian classes. Thin contours show each class density. Left: equal covariances give a straight line. Right: unequal covariances give a curved (quadratic) boundary that wraps around the tighter class.

Estimating $\mathbf{m}_j$ and $\mathbf{C}_j$ from samples is easy: the sample mean and sample covariance. The priors can be the class frequencies in the training set, if those reflect reality.

```
import numpy as np

class GaussianBayes:
    def fit(self, X, y):
        self.classes = np.unique(y)
        self.m = [X[y == c].mean(0) for c in self.classes]
        self.C = [np.cov(X[y == c].T) for c in self.classes]
        self.logP = [np.log((y == c).mean()) for c in self.classes]
        return self

    def decision(self, X):
        scores = []
        for m, C, lp in zip(self.m, self.C, self.logP):
            d = X - m
            maha = np.einsum("ni,ij,nj->n", d, np.linalg.inv(C), d)
            scores.append(lp - 0.5 * np.linalg.slogdet(C)[1] - 0.5 * maha)
        return np.stack(scores, axis=1)            # (n, W): one d_j(x) per class

    def predict(self, X):
        return self.classes[self.decision(X).argmax(1)]
```

The number of covariance parameters grows as $n(n+1)/2$ per class. With 4 features that is 10 numbers per class, easily estimated. With 1,000 features it is about half a million, and the estimate

becomes unreliable unless there is a huge amount of data. This is one reason the Gaussian Bayes classifier works best on short, well-chosen feature vectors.

Application: classifying pixels in a multispectral image

Remote-sensing satellites record each ground location in several spectral bands. Each pixel is then a natural pattern vector, $\mathbf{x} = (x_{\text{blue}}, x_{\text{green}}, x_{\text{red}}, x_{\text{NIR}})^{\top}$, where NIR is near-infrared. Healthy vegetation reflects strongly in NIR and weakly in red; water is dark in NIR; built-up areas are moderately bright everywhere.

We built a synthetic 128×128 scene with three classes (water, vegetation, built-up), gave each class a mean reflectance in four bands, and added independent Gaussian noise with standard deviation 0.06 to every band. We picked only 50 random pixels per class as training data, estimated $\mathbf{m}_j$ and $\mathbf{C}_j$, and classified every pixel with the Gaussian Bayes rule.

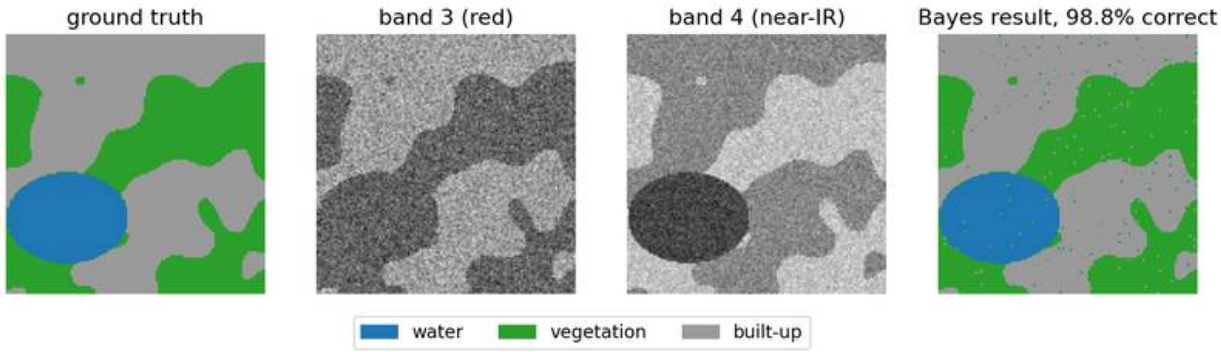


Figure 13.4 — Bayes classification of a synthetic four-band scene. Single bands are noisy and ambiguous (red barely separates water from vegetation), but the four-dimensional pattern vector separates the classes well: 98.8% of pixels are labelled correctly from only 150 training pixels.

Two lessons stand out. First, no single band is enough: in the red band, water and vegetation have similar values. The **combination** of bands separates them. Second, the remaining errors are isolated pixels. The Bayes rule here treats each pixel independently and ignores neighbors. Adding spatial context, for example by smoothing the decision scores or using a classifier that sees a neighborhood, removes most of those errors. That observation leads naturally to convolutional networks.

Neural networks and deep learning

Plain version. A neural network is a stack of very simple units. Each unit adds up its inputs with weights, then decides how strongly to "fire". By nudging the weights after each mistake, the network learns a decision rule on its own, without our writing down the probability distributions.

Background

The Bayes classifier needs density estimates, which are hard to get in high dimensions. An alternative is to **learn the decision functions directly** from training data. Neural networks do this with many simple, adjustable computing elements arranged in layers. The idea goes back to the perceptron of the late 1950s [4]. It became practical for multilayer networks once backpropagation was popularized in 1986 [5], and it became dominant for images after deep convolutional networks won the ImageNet challenge in 2012 [7]. For a fuller treatment of perceptrons, MLPs, activation functions and training tricks, see ML 02 ·Neural Networks.

The perceptron

A **perceptron** computes a weighted sum of its inputs plus a bias and outputs the sign:

$$d(\mathbf{x}) = \mathbf{w}^{\top}\mathbf{x} + w_0 = \sum_{k=1}^{n} w_k x_k + w_0, \qquad \text{class} = \begin{cases} \omega_1 & d(\mathbf{x}) > 0 \\ \omega_2 & d(\mathbf{x}) \le 0 \end{cases}$$

where $\mathbf{w} = (w_1, \dots, w_n)^{\top}$ are the **weights** and w_0 is the **bias**. The boundary $d(\mathbf{x}) = 0$ is a hyperplane. To simplify notation, append a constant 1 to every pattern, $\mathbf{y} = (x_1, \dots, x_n, 1)^{\top}$, and fold the bias into the weights, $\mathbf{w} = (w_1, \dots, w_n, w_0)^{\top}$, so that $d = \mathbf{w}^{\top}\mathbf{y}$.

Learning rule. Encode the label as $t = +1$ for ω_1 and $t = -1$ for ω_2. Present the training patterns one at a time. At step k:

$$\mathbf{w}(k+1) = \begin{cases} \mathbf{w}(k) + \alpha\, t\, \mathbf{y}(k) & \text{if } t\, \mathbf{w}(k)^{\top}\mathbf{y}(k) \le 0 \text{ (misclassified)} \\ \mathbf{w}(k) & \text{otherwise,} \end{cases}$$

where $\alpha > 0$ is the **learning rate**. Each correction moves the hyperplane so that the offending pattern is pushed toward its correct side: after the update, $t\,\mathbf{w}^{\top}\mathbf{y}$ increases by $\alpha\|\mathbf{y}\|^2$. If the two classes are **linearly separable**, this procedure is guaranteed to stop after a finite number of corrections with a separating hyperplane. If they are not, it never settles. In that case we instead minimize a smooth error, such as the squared difference between $\mathbf{w}^{\top}\mathbf{y}$ and t, by gradient descent (the least-mean-squares, or delta, rule). That idea leads directly to backpropagation.

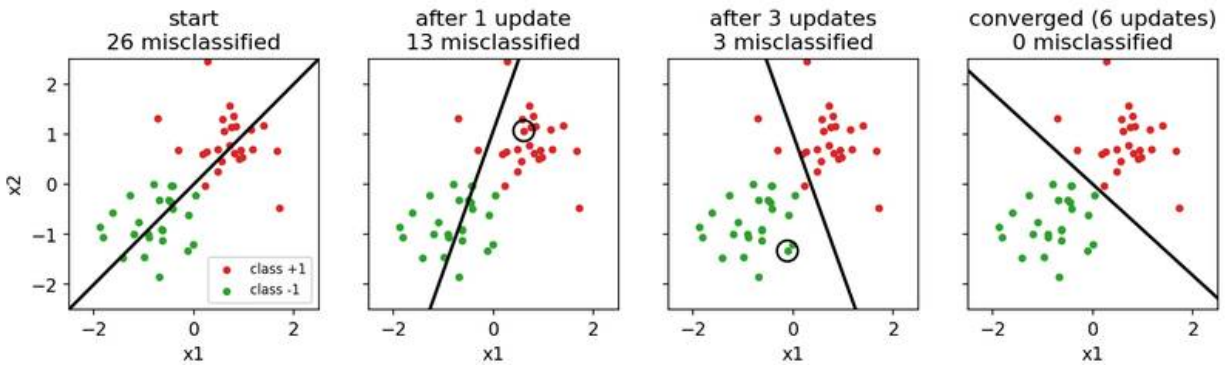


Figure 13.5 — Perceptron training on two separable classes. The circled point is the misclassified pattern that triggered the update. After six corrections the line separates the classes and learning stops.

```
import numpy as np

def train_perceptron(X, y, alpha=0.5, epochs=100):
    """X: (n, d), y in {-1, +1}. Returns augmented weights (w_1..w_d, w_0)."""
    Xa = np.c_[X, np.ones(len(X))]                # append the constant input 1
    w = np.zeros(Xa.shape[1])
    for _ in range(epochs):
        mistakes = 0
        for xi, yi in zip(Xa, y):
            if yi * (w @ xi) <= 0:                # wrong side (or on the line)
                w += alpha * yi * xi
                mistakes += 1
        if mistakes == 0:                         # a full clean pass: converged
            break
    return w
```

A single perceptron can only draw one hyperplane. It cannot solve XOR, where the two classes sit on opposite corners of a square. To draw curved or multi-piece boundaries we need more layers.

Multilayer feedforward networks

A **multilayer feedforward network** (or multilayer perceptron, MLP) stacks layers of units. Layer ℓ receives the outputs of layer $\ell - 1$, and nothing flows backward during prediction. Unlike the perceptron, each unit passes its weighted sum through a smooth, differentiable **activation function** h, such as the sigmoid $h(z) = 1/(1 + e^{-z})$, the hyperbolic tangent, or the rectified linear unit $\mathrm{ReLU}(z) = \max(0, z)$. Smoothness is what makes gradient-based training possible. Nonlinearity is what makes depth useful: a stack of purely linear layers collapses into one linear layer.

With two hidden layers and enough units, a network can form decision regions of essentially any shape. In practice the question is not whether a network **can** represent a boundary, but whether training will **find** it from the data available.

Forward pass

In plain words: each layer multiplies its input by a weight matrix, adds a bias and applies the activation; the last layer turns its scores into class probabilities.

Number the layers $\ell = 1, \dots, L$, with layer 0 being the input $\mathbf{a}^{(0)} = \mathbf{x}$. For each layer,

$$\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}, \qquad \mathbf{a}^{(\ell)} = h(\mathbf{z}^{(\ell)}),$$

where $\mathbf{W}^{(\ell)}$ is the weight matrix (row i holds the weights into unit i), $\mathbf{b}^{(\ell)}$ the bias vector, $\mathbf{z}^{(\ell)}$ the **net inputs** and $\mathbf{a}^{(\ell)}$ the **activations**, with h applied element by element. For a W-class problem the output layer has W units. Its net inputs are turned into probabilities with the **softmax** function,

$$p_j = \frac{e^{z_j^{(L)}}}{\sum_{i=1}^{W} e^{z_i^{(L)}}}, \qquad j = 1, \ldots, W,$$

and the predicted class is the one with the largest p_j. Notice that the output layer is a set of decision functions $d_j(\mathbf{x}) = z_j^{(L)}$, exactly as in the Background section. The hidden layers compute features in which those simple, linear decision functions suffice.

Backpropagation

Training adjusts all weights to reduce a **loss** E that measures how wrong the outputs are. For classification the standard choice is the **cross-entropy** $E = -\sum_j t_j \ln p_j$, where $\mathbf{t}$ is the one-hot label (1 for the true class, 0 elsewhere). For a sigmoid output the classic alternative is the squared error $E = \frac{1}{2}\|\mathbf{a}^{(L)} - \mathbf{t}\|^2$. We need $\partial E / \partial \mathbf{W}^{(\ell)}$ for every layer. Backpropagation computes all of them with one backward sweep of the chain rule [5].

In plain words: backpropagation sends the output error backward through the network, layer by layer, and the gradient of each weight is "the error arriving at its output end" times "the activation at its input end". Define the **error signal** of layer ℓ as $\delta^{(\ell)} = \partial E / \partial \mathbf{z}^{(\ell)}$. For softmax with cross-entropy the output signal is simply $\delta^{(L)} = \mathbf{p} - \mathbf{t}$. Each earlier signal is the next one passed back through the transposed weights and multiplied by the activation's derivative; the gradient of $\mathbf{W}^{(\ell)}$ is $\delta^{(\ell)} \mathbf{a}^{(\ell-1)\top}$, and gradient descent then updates every weight with a learning rate α. The backward pass costs about as much as the forward pass, which is why networks with millions of weights can be trained at all. The perceptron learning rule [4] is a separate mistake-driven update, not ordinary backpropagation through a hard step: the step derivative is zero away from its threshold and undefined at the threshold. Differentiable activations let the chain rule in [5] supply useful gradients.

Online only: Backpropagation step by step — https://blog.shyandram.dev/en/learn/image-processing/pattern-classification/notes/backpropagation-step-by-step/

Training

In practice we average the gradient over a small random **mini-batch** of training patterns, update, and repeat. This is **stochastic gradient descent** (SGD). One pass through the whole training set is an **epoch**. Training runs for many epochs while we watch the loss on the validation set. The learning rate is the most important setting: too large and the loss oscillates or explodes; too small and training crawls. Weights must start as small random numbers, not zeros, or all units in a layer would compute the same thing and receive the same updates forever.

The following NumPy program trains a two-layer network on XOR, the problem a single perceptron cannot solve:

```
import numpy as np

rng = np.random.default_rng(0)
```

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], float)
y = np.array([[0], [1], [1], [0]], float)         # XOR: not linearly separable

W1, b1 = rng.normal(0, 1, (2, 8)), np.zeros(8)    # input -> 8 hidden units
W2, b2 = rng.normal(0, 1, (8, 1)), np.zeros(1)    # hidden -> 1 output
sig = lambda z: 1 / (1 + np.exp(-z))
alpha = 1.0

for step in range(5000):
    # forward pass
    a1 = np.tanh(X @ W1 + b1)
    out = sig(a1 @ W2 + b2)
    # backward pass (cross-entropy loss with a sigmoid output)
    d2 = (out - y) / len(X)                        # delta at the output layer
    d1 = (d2 @ W2.T) * (1 - a1 ** 2)               # delta at the hidden layer
    W2 -= alpha * a1.T @ d2;  b2 -= alpha * d2.sum(0)
    W1 -= alpha * X.T @ d1;   b1 -= alpha * d1.sum(0)

print(out.round(3).ravel())                        # close to [0, 1, 1, 0]
```

(Here patterns are rows, so the matrix products are transposed relative to the equations above.)

Deep convolutional neural networks

Plain version. Instead of telling the network which features to measure, we let it slide small learnable filters over the image, exactly like the filters of IP 03, and learn which filters help it classify. Early layers end up finding edges and blobs; later layers combine them into parts and whole objects.

Why fully connected networks struggle with images

Feed a 20×20 image into a fully connected layer with 2,048 hidden units, and that one layer has $400 \times 2048 = 819{,}200$ weights. The layer also ignores the fact that nearby pixels are related and that a "3" shifted two pixels to the right is still a "3". A CNN builds both facts into its structure [6].

Convolutional layers

A **convolutional layer** applies F small kernels (for example 5×5) to its input. With C input channels a_c, the net input of output channel f at pixel (x, y) is

$$z_f(x,y) = \sum_{c=1}^{C} \sum_{s,t} w_{f,c}(s,t)\, a_c(x+s, y+t) + b_f,$$

followed by an activation, usually ReLU: $a_f(x,y) = \max(0, z_f(x,y))$. The sum is a spatial correlation, as in IP 03, but the kernel values $w_{f,c}$ are **learned**. Each output channel is a **feature map**: it is large where its kernel's pattern appears. Two ideas cut the parameter count dramatically:

- **Local connectivity.** Each output depends only on a small neighborhood (its **receptive field**).
- **Weight sharing.** The same kernel is used at every position, so a feature learned in one place is detected everywhere.

Eight 5×5 kernels on a single-channel image need $8\times25+8 = 208$ parameters, against the 819,200 above.

Pooling layers

A **pooling layer** shrinks each feature map, typically by taking the maximum (max-pooling) or average of each non-overlapping 2×2 block. It halves the width and height, makes the response tolerant to small shifts, and enlarges the receptive field of later layers. Stacking convolution and pooling several times yields features that respond to larger and more abstract structures.

Fully connected layers and the whole pipeline

After the last pooling layer the feature maps are **flattened** into one vector and passed through one or more fully connected layers ending in a softmax. Figure 13.6 shows the small network we trained.

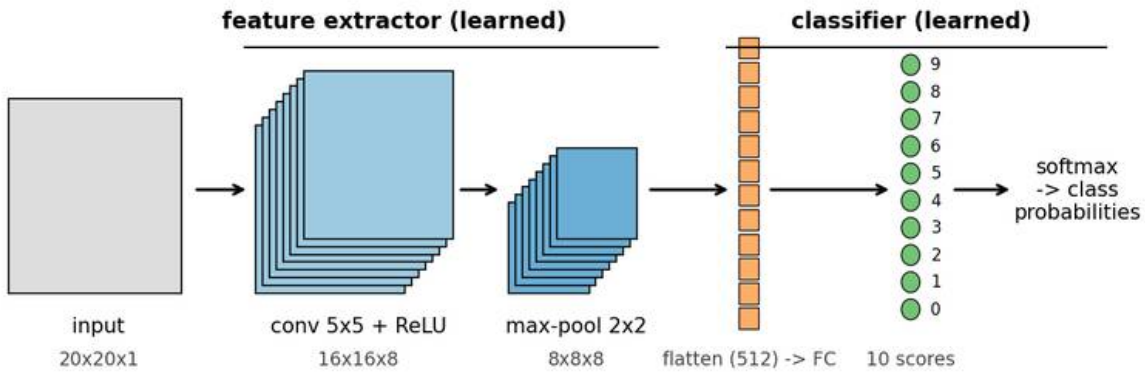


Figure 13.6 — The tiny CNN used in this chapter. The convolution and pooling layers form a learned feature extractor; the fully connected layer is a learned linear classifier on those features.

This is the key conceptual point of the chapter. The classical pipeline was **hand-designed features → classifier**. A CNN is **learned features → classifier**, with both parts trained together by backpropagation to minimize the same loss. The fully connected softmax layer at the end is just a set of linear decision functions, the same object we met in the minimum-distance and equal-covariance Bayes classifiers. What changed is that the space in which those linear functions operate is itself learned.

Training a CNN

Backpropagation works unchanged; only the local derivatives differ. A shared kernel weight collects its gradient from every position where it was used, max-pooling passes the gradient only to the input that was the maximum, and ReLU passes it only where its input was positive.

Online only: Gradients through convolution, pooling and ReLU — https://blog.shyandram.dev/en/learn/image-processing/pattern-classification/notes/gradients-through-convolution-pooling-and-relu/

Here is the forward pass of one convolution, ReLU and pooling stage with SciPy:

```
import numpy as np
from scipy.signal import correlate2d
from skimage import data, transform

img = transform.resize(data.camera(), (64, 64), anti_aliasing=True)
k = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], float)    # one 3x3 "filter"
fmap = np.maximum(correlate2d(img, k, mode="valid") + 0.0, 0)  # conv + bias + ReLU
pooled = fmap[:62, :62].reshape(31, 2, 31, 2).max(axis=(1, 3))  # 2x2 max-pooling
print(img.shape, "->", fmap.shape, "->", pooled.shape)
```

Worked example: digits

The classic test bed for CNNs is handwritten digit recognition, where LeNet-style networks were developed on the MNIST digits [6]. To keep everything self-contained and reproducible, we generate **MNIST-like** 20 × 20 digits: each digit 0–9 is drawn with one of six OpenCV fonts at a random size, stroke width, shift and rotation ($\pm 15°$), then noise is added (Figure 13.7).

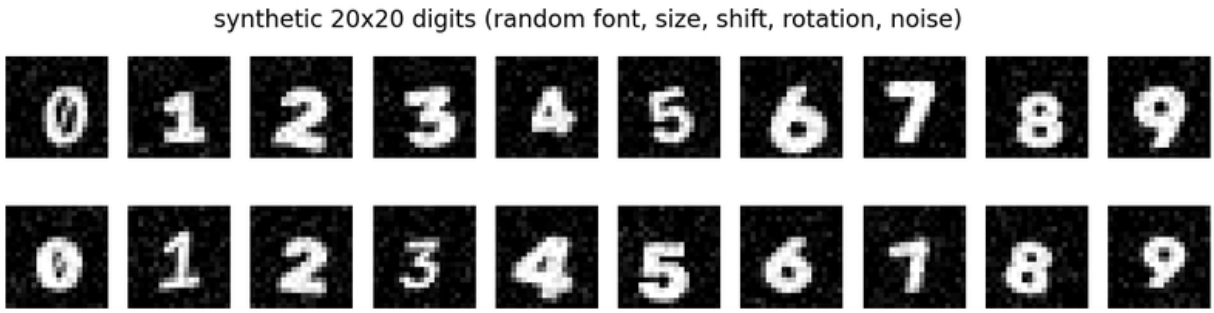


Figure 13.7 — Synthetic training digits. Variation in font, size, position and angle forces the classifier to learn shape rather than memorize pixels.

The network of Figure 13.6 (eight 5×5 filters, ReLU, 2×2 max-pooling, a fully connected layer to 10 outputs, softmax with cross-entropy) is written in about 80 lines of NumPy in `scripts/figures/dip_ch13.py`. We trained it with mini-batch SGD (batch size 32, learning rate 0.05, small weight decay) on 3,000 digits for 12 epochs. It reached **98.5%** accuracy on 1,000 separate validation digits. Figure 13.8 shows what it learned.

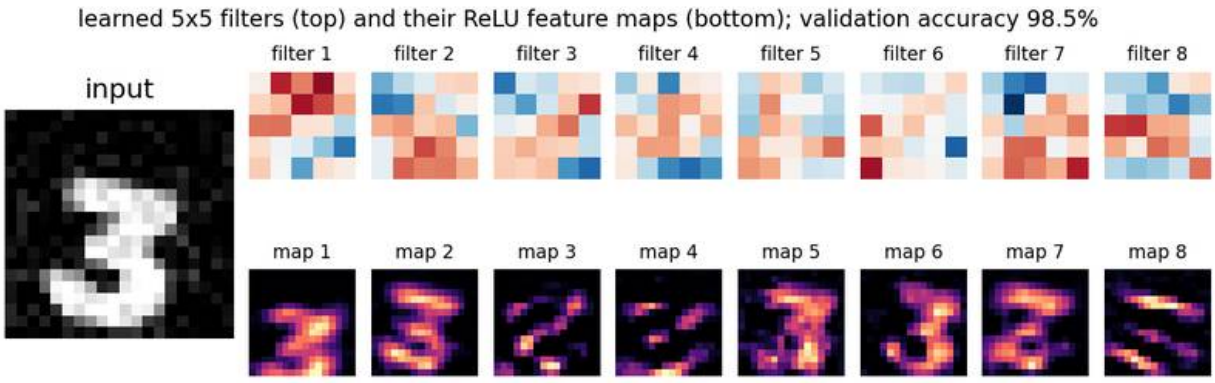


Figure 13.8 — Inside the trained CNN. Top: the eight learned 5×5 kernels (red positive, blue negative). Bottom: their ReLU feature maps for the input digit. Several kernels act as oriented edge or stroke detectors, similar to the hand-designed gradient filters of IP 10.

Nobody told the network to detect edges. Oriented, edge-like kernels emerge because they are useful for telling digits apart. Large networks trained on natural images show the same behavior in their first layer, with progressively more complex patterns in deeper layers.

Some additional details of implementation

Plain version. A network that memorizes its homework can still fail the exam. Most of the practical craft of training is about making it learn the general rule instead of the specific examples.

Overfitting

A network with many parameters and few training examples can drive its training error to zero by memorizing. Its error on new data then stays high. This gap between training and validation performance is **overfitting**. Figure 13.9 (left) shows it clearly: trained on only 150 digits, the CNN reaches 100% training accuracy but only 86.2% validation accuracy.

The basic defenses:

- **More data.** Nothing helps as reliably. Data augmentation (below) is a cheap way to get more.
- **A validation set and early stopping.** Track validation accuracy and keep the weights from the best epoch.
- **Smaller or better-structured models.** Weight sharing in CNNs is itself a strong regularizer compared with fully connected layers.

Data augmentation

Data augmentation creates new training examples by applying label-preserving transformations: small shifts, rotations, scaling, flips (where they make sense: a flipped "cat" is still a cat, but a

flipped "b" becomes a "d"), crops, brightness and contrast changes, and added noise [18]. Each epoch sees a slightly different version of each image, so memorizing is harder and invariance to those transformations is learned from data.

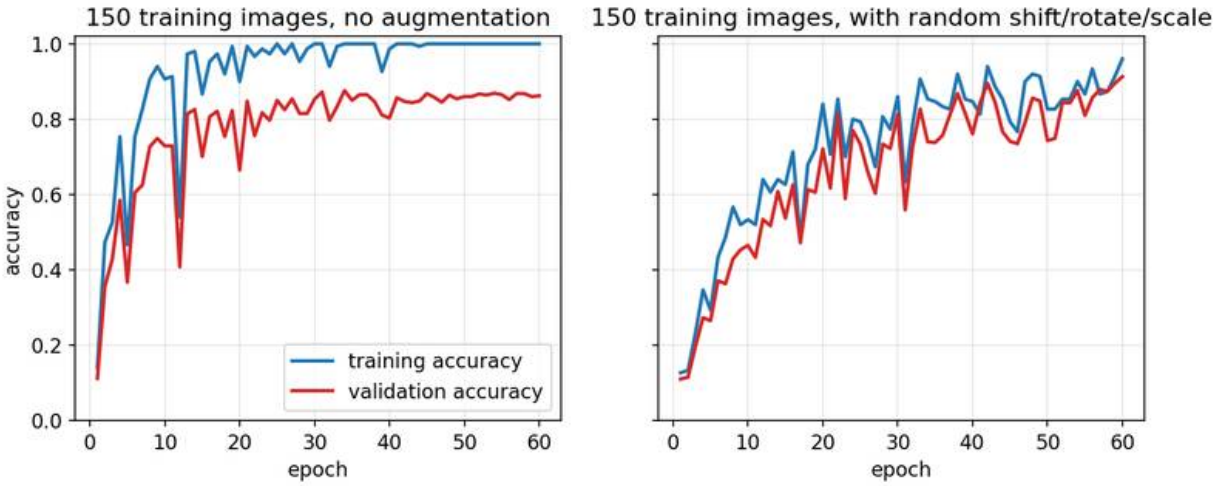


Figure 13.9 — The same CNN trained on only 150 synthetic digits for 60 epochs. Left: without augmentation, a large train–validation gap opens (overfitting). Right: random shifts, rotations and scaling keep the curves together, and final validation accuracy rises from 86.2% to 91.3%.

Regularization

Regularization adds a preference for simpler solutions:

- **Weight decay** (L_2 regularization) adds $\frac{\lambda}{2}\sum w^2$ to the loss, which adds λw to every weight's gradient and shrinks weights toward zero. Our training script uses $\lambda = 10^{-4}$.
- **Dropout** randomly sets a fraction of the units to zero at each training step, so no unit can rely on specific partners. At test time all units are used, with appropriately scaled weights. This behaves like averaging an ensemble of many thinned networks [17].

Other practical details matter just as much: normalizing inputs (zero mean, unit variance), initializing weights with a variance scaled to the number of inputs (so activations neither explode nor vanish through the layers), lowering the learning rate during training, and always reporting final results on a test set that was not used for any decision during development.

Modern view

The textbook chapter ends where modern image classification begins. Six survey and review papers map how the field has moved since.

Statistical pattern recognition before deep learning. Jain, Duin and Mao's review [2] maps the classical territory, from representation and feature selection to classifiers, classifier combina-

tion and error estimation. Its main takeaway, that the feature representation and the evaluation protocol matter more than the choice of classifier, aged well: deep learning automated the representation part.

The CNN era. Rawat and Wang [11] review CNNs for image classification up to 2017. The line runs from LeCun et al.'s gradient-trained networks for document recognition [6] through AlexNet [7], winner of the ImageNet challenge in 2012 [10], to VGG [8] and ResNet [9], whose identity shortcuts made very deep networks trainable. Each is still recognizably the architecture of Figure 13.6.

Transformers in vision. The Vision Transformer (ViT) [12] treats image patches as tokens and replaces convolution with self-attention; Khan et al. [13] survey its many variants, and ConvNeXt [14] showed that a modernized pure CNN is competitive again. The practical lesson: architecture families matter less than training data, scale and recipe.

Foundation features and zero-shot classification. CLIP [15] trains an image encoder and a text encoder together, so a classifier can be built without labelled images by comparing each image with text prompts such as `a photo of a {class}`: a **minimum-distance classifier** whose prototypes come from language. DINOv2 [16] learns general-purpose features by self-supervision, and a linear layer or a nearest-neighbor rule on its frozen features classifies well. The expensive feature extractor is trained once; the classifier on top is often one of this chapter's simple rules.

Data augmentation. Shorten and Khoshgoftaar's survey [18] sorts augmentation methods into basic image manipulations and learned approaches. Its takeaway matches Figure 13.9: augmentation is one of the cheapest and most effective ways to reduce overfitting when labelled data are limited.

Evaluation pitfalls and dataset bias. Torralba and Efros [19] showed that a classifier can tell which dataset an image came from, and Liu and He [20] found that this still holds for modern networks and web-scale datasets. Accuracy on one benchmark is a statement about that benchmark; whenever possible, test on data collected independently of the training data.

Online only: Guide to the Modern view papers — https://blog.shyandram.dev/en/learn/image-processing/pattern-classification/papers/guide-to-the-modern-view-papers/

What changed and what did not. Deep learning replaced hand-crafted features and made the classifier itself almost an afterthought: a linear softmax layer. It did not change the decision-theoretic foundations. The softmax output is trained to estimate posterior probabilities, so the network approximates the Bayes rule. Cross-entropy is the negative log-likelihood. Prototype matching survives as nearest-neighbor search in embedding spaces, used in zero-shot classification and retrieval. Template matching by normalized correlation is still the right tool in controlled industrial inspection, where the object's appearance is fixed and training data are scarce. Understanding the classical methods tells you what the modern ones are optimizing and when a simpler method is enough.

Key takeaways

- Classification assigns a label to a pattern via decision functions $d_j(\mathbf{x})$; the boundaries are where two decision functions are equal.
- Minimum-distance classification uses class means as prototypes; its boundaries are perpendicular bisectors. Normalized correlation and SIFT matching are prototype matching for images and keypoints.
- The Bayes classifier minimizes the average loss. For Gaussian classes it is quadratic; with equal covariances it is linear; with identity covariance and equal priors it reduces to minimum distance.
- A perceptron learns a separating hyperplane by correcting mistakes. Multilayer networks with smooth activations learn nonlinear boundaries, and backpropagation computes all gradients in one backward pass.
- A CNN is a learned feature extractor (convolution, nonlinearity, pooling) followed by a learned linear classifier. Weight sharing makes it efficient and shift-tolerant.
- Overfitting is the central practical risk. Use validation data, data augmentation, weight decay and dropout, and keep the test set untouched.
- Modern systems often pair a large pre-trained feature extractor (CLIP, DINOv2) with a very simple classifier. Benchmark accuracy reflects dataset biases, so test on independently collected data.

Exercises

1. **Bisector by hand.** Two class means are $\mathbf{m}_1 = (1,2)^\top$ and $\mathbf{m}_2 = (5,4)^\top$. Write the minimum-distance boundary $d_1(\mathbf{x}) - d_2(\mathbf{x}) = 0$ as an equation in x_1, x_2, and verify that the midpoint of the two means lies on it.

   Hint

   $\mathbf{m}_1 - \mathbf{m}_2 = (-4,-2)^\top$ and
   $\frac{1}{2}(\mathbf{m}_1 - \mathbf{m}_2)^\top(\mathbf{m}_1 + \mathbf{m}_2) = \frac{1}{2}(-4 \cdot 6 - 2 \cdot 6) = -18$. The boundary is $-4x_1 - 2x_2 + 18 = 0$, i.e. $2x_1 + x_2 = 9$. The midpoint $(3,3)$ gives $6 + 3 = 9$. ✓

2. **When priors matter.** In a one-dimensional problem, class ω_1 is Gaussian with mean 0 and class ω_2 with mean 4, both with variance 1. Find the Bayes threshold when $P(\omega_1) = P(\omega_2)$, and when $P(\omega_1) = 0.9$. In which direction does the threshold move, and why does that make sense?

   Hint

   Set $\ln P_1 - \frac{1}{2}x^2 = \ln P_2 - \frac{1}{2}(x-4)^2$. This gives $x = 2 + \frac{1}{4}\ln(P_1/P_2)$. Equal priors give $x = 2$; $P_1 = 0.9$ gives $x = 2 + \frac{1}{4}\ln 9 \approx 2.55$. The threshold moves toward the rarer class, so more ambiguous patterns go to the common class.

3. **Correlation invariance.** Show that the normalized correlation coefficient γ is unchanged if every pixel of the image patch is replaced by $af + b$ with $a > 0$. What happens if $a < 0$? Then test it with `cv2.matchTemplate` on `skimage.data.camera()`.

   Hint

   Subtracting the mean removes b. The factor a appears once in the numerator and as $\sqrt{a^2} = |a|$ in the denominator, so it cancels for $a > 0$. For $a < 0$ (a photographic negative) the sign of γ flips: a perfect match becomes -1.

4. **A perceptron that never stops.** Generate two overlapping Gaussian clouds and run the perceptron rule from this chapter for 100 epochs. Plot the number of mistakes per epoch. Then replace the rule by gradient descent on the squared error $\frac{1}{2}\sum(\mathbf{w}^\top \mathbf{y} - t)^2$ and compare the two learning curves.

   Hint

   The perceptron's mistake count keeps fluctuating, because every mistake moves the line, even when no line can be perfect. The least-squares rule converges to a fixed compromise line, since its loss is smooth and convex.

5. **Counting parameters.** A network takes 64×64 grayscale images. Compare the number of weights (including biases) in (a) a fully connected layer with 1,024 hidden units and (b) a convolutional layer with 32 kernels of size 3×3. How many outputs does (b) produce without padding?

   Hint

   (a) $4096 \times 1024 + 1024 = 4{,}195{,}328$. (b) $32 \times 9 + 32 = 320$ parameters, producing 32 maps of 62×62, i.e. 123,008 outputs. Few parameters, many outputs: that is weight sharing.

6. **Augmentation that hurts.** Using the digit generator in `scripts/figures/dip_ch13.py`, add random 180° rotation to the augmentation. Retrain and inspect the confusion between which digits rises the most. Explain why.

   Hint

   A "6" rotated by 180° looks like a "9" (and "0", "1", "8" look like themselves). The augmentation is no longer label-preserving, so the network is taught contradictory labels for the same shape. Expect the 6/9 confusion to rise sharply.

References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018, Ch. 13. publisher page
2. A. K. Jain, R. P. W. Duin, and J. Mao, "Statistical Pattern Recognition: A Review," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 1, pp. 4–37, 2000. doi:10.1109/34.824819
3. D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, 2004. doi:10.1023/B:VISI.0000029664.99615.94
4. F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychological Review*, vol. 65, 1958. doi:10.1037/h0042519
5. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-Propagating Errors," *Nature*, vol. 323, 1986. doi:10.1038/323533a0
6. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, 1998. doi:10.1109/5.726791
7. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems 25* (*NIPS*), 2012. paper
8. K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," arXiv:1409.1556, 2014. arXiv
9. K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," arXiv:1512.03385, 2015. arXiv
10. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, et al., "ImageNet Large Scale Visual Recognition Challenge," arXiv:1409.0575, 2014. arXiv
11. W. Rawat and Z. Wang, "Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review," *Neural Computation*, vol. 29, no. 9, 2017. doi:10.1162/neco_a_00990
12. A. Dosovitskiy, L. Beyer, A. Kolesnikov, et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *ICLR*, 2021. arXiv
13. S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah, "Transformers in Vision: A Survey," *ACM Computing Surveys*, 2022. arXiv
14. Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A ConvNet for the 2020s," *CVPR*, 2022. arXiv
15. A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, et al., "Learning Transferable Visual Models From Natural Language Supervision," arXiv:2103.00020, 2021. arXiv
16. M. Oquab, T. Darcet, T. Moutakanni, et al., "DINOv2: Learning Robust Visual Features without Supervision," arXiv:2304.07193, 2023. arXiv
17. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. JMLR
18. C. Shorten and T. M. Khoshgoftaar, "A Survey on Image Data Augmentation for Deep Learning," *Journal of Big Data*, vol. 6, 2019. doi:10.1186/s40537-019-0197-0
19. A. Torralba and A. A. Efros, "Unbiased Look at Dataset Bias," *CVPR*, 2011. paper page
20. Z. Liu and K. He, "A Decade's Battle on Dataset Bias: Are We There Yet?," *ICLR*, 2025. arXiv