Machine Vision Algorithms: Principles and Industrial Practice

Will Pan

Table of contents

Preface

This book systematically covers the algorithmic principles of industrial machine vision: from imaging and calibration, image enhancement, locating, measurement, inspection, and recognition to 3D imaging and 3D processing. Each chapter starts with intuition and mathematical principles, accompanied by pseudocode and runnable SciVision SDK C++ examples, and presents "Industry Case" callouts with real parameter experience from production lines.

The companion code for the book lives in the repository's `code/` directory.

About the Author

Will Pan is a researcher in machine vision and 3D imaging — industrial 3D imaging, point-cloud and mesh processing, measurement and inspection, and robot vision. Ph.D. from the Singapore University of Technology and Design (SUTD).

Notation

Symbol	Meaning
$f[n, m]$	Discrete image, n is the row index (downward), m is the column index (rightward)
$g[n, m]$	Output image
$h[n, m]$	Convolution kernel / filter
$\circ$	Convolution operation
σ	Gaussian standard deviation or noise standard deviation (depending on context)
$\mathbf{x} = (x, y)^\top$	Continuous coordinate point
θ	Angle (degrees or radians, stated by context)
K	Kernel/window size (e.g., $K \times K$)
κ	Knee threshold of the Huber weight

Part I

Foundations

This part establishes the common language of the book: the representation, sampling, and quantization of digital images, together with mathematical preliminaries in linear algebra, probability and statistics, and optimization that underpin all later parts.

1 Digital Image Fundamentals

At the instant the lens focuses light onto the sensor's active surface, the "image" is still a continuous physical quantity: every point on the image plane has its own illuminance, and brightness can take any real value. Yet what the image-processing program on the industrial PC receives is a matrix of integers. Between the former and the latter, two discretizations take place inside the camera: **sampling** — the sensor's photosite array chops the continuous image plane into a finite grid of points, determining how finely the image resolves space; and **quantization** — the analog-to-digital converter compresses the continuous charge on each photosite into an integer with a finite number of levels, determining how finely the image resolves gray values. Each step throws away part of the information. The question this chapter answers is precisely this: when does the lost information not matter, and when does it come back to wreck an entire vision solution?

So that the consequences of both discretizations can be seen with the naked eye, this chapter constructs a 480×360, 8-bit grayscale test scene (Figure 1.1). A horizontal linear gradient background running from 40 to 160 and a Lambertian sphere provide **smooth, continuous gray values** — these are the most sensitive to quantization. A set of nested step rectangles (a dark 20 enclosing a bright 235) provides **sharp edges**. Groups of vertical bars 1, 2, and 3 pixels wide, together with a chirp stripe band whose frequency rises continuously from left to right, provide **fine structures** — these are the most sensitive to sampling. Every experimental figure that follows is genuinely generated by the programs under `code/digital_images/`.

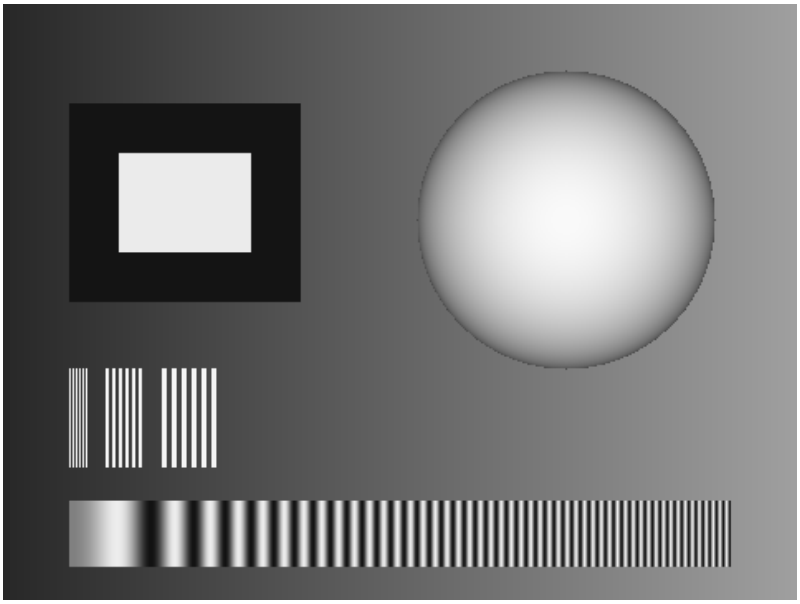


Figure 1.1: The baseline test scene (480×360, 8-bit). Top left: a bright rectangle nested inside a dark one (step edges); top right: a Lambertian sphere (a continuous gray-value surface); the background is a horizontal gradient from 40 to 160; bottom left: groups of vertical bars 1/2/3 px wide; bottom: a chirp stripe band whose frequency increases to the right.

1.1 The Image as a Two-Dimensional Signal

Write the continuous image projected by the lens onto the sensor as $f_c(x, y)$. Sampling turns it into a **discrete image** defined only on an integer grid,

$$f[n, m] = f_c(m\Delta_x,\ n\Delta_y),$$

where n is the row index (increasing downward), m is the column index (increasing to the right), and Δ_x, Δ_y are the sampling intervals — physically, the pitch of the photosites. Quantization then restricts each value $f[n, m]$ to a finite set of integer levels. A digital image is thus, in the end, a matrix of integers; the test scene of this chapter is exactly 360 rows $\times$ 480 columns.

This brings out the three most fundamental specifications. **Resolution** is the size of the sampling grid — the image's width and height in pixels — and it sets the upper limit on spatial detail. **Bit depth** is the number of binary bits used to represent each pixel, which determines the number of gray levels — 8 bits give $2^8 = 256$ levels, the most common output format for industrial cameras. **Dynamic range** is the ratio between the brightest and darkest signals the sensor can record simultaneously; bit depth determines into how many levels that range is divided for representation. The three are mutually independent: high resolution does not imply high bit depth, and a high bit depth does not mean the sensor can actually distinguish that many effective gray levels.

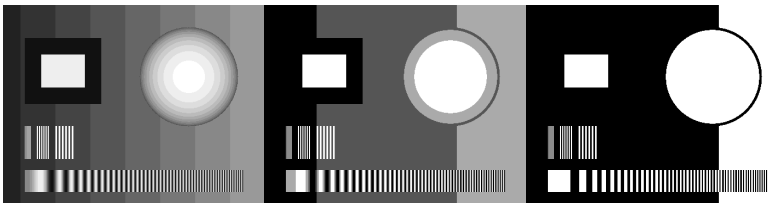
1.2 Quantization

Requantizing 256 gray levels down to L levels follows a simple rule: cut the gray axis into L segments of step size $\Delta = 256/L$, and record a pixel as level q if its value falls in the q-th segment:

Common bit depths for industrial cameras are 8/10/12 bits. For most locating and inspection tasks, 8 bits is enough; but when an algorithm uses gray values as a **physical quantity** — for example, photometric stereo (Chapter 34) recovering surface normals from shading, or measuring the brightness uniformity of a backlight — the 256 levels of 8 bits are often insufficient, and 10-bit or even 12-bit output is required.

$$q[n,m] = \left\lfloor \frac{f[n,m]}{\Delta} \right\rfloor, \qquad \Delta = \frac{256}{L}.$$

The error introduced by quantization is at most half a step, $\pm\Delta/2$, and is approximately uniformly distributed within each step — you can regard it as a kind of uniform noise superimposed on the image. It belongs to the same family of "perturbed gray values" as the sensor noise discussed in Chapter 6, except that it is determined by our own digitization and cannot be removed by averaging. The smaller L, the larger the step, and the stronger this "noise". Figure 1.2 shows the same scene quantized to 16 levels (4-bit, step 16), 4 levels (2-bit, step 64), and 2 levels (1-bit, step 128).



(a) 16 levels (4-bit) (b) 4 levels (2-bit) (c) 2 levels (1-bit)

Figure 1.2: The gray-level quantization experiment. (a) 16 levels: vertical bands appear across the gradient background and concentric rings on the sphere, while the rectangle edges and the bar groups are almost unaffected; (b) 4 levels: severe posterization — only four shades of gray remain; (c) 2 levels: pure black and white — the sphere merges into the bright background on the right, and the sphere's outline shrinks to a thin ring.

The first place quantization shows itself is not the edges but the **smooth gradient regions**. In Figure 1.2a, the gradient background breaks into clearly visible vertical **banding** artifacts, and the sphere turns into a set of concentric rings. The reason is not hard to see: the background climbs from 40 to 160 across 480 columns, so the gray-value difference between adjacent pixels is far smaller than the quantization step of 16. The continuous ramp is therefore sliced into a staircase of constant-gray plateaus, with a jump of a full step at each plateau boundary

— and the human eye is extremely sensitive to large-area gray steps, so the banding is unmistakable. The sphere is the two-dimensional version of the same phenomenon: its constant-gray plateaus are ring-shaped, hence the concentric rings. Pressing on to 4 levels (Figure 1.2b), the entire image is reduced to four shades of gray — classic **posterization**. At 2 levels (Figure 1.2c) only a single threshold survives: most of the sphere and the brighter background on the right are both assigned to "white" and fuse into one region, and the sphere's shape as an object is essentially lost.

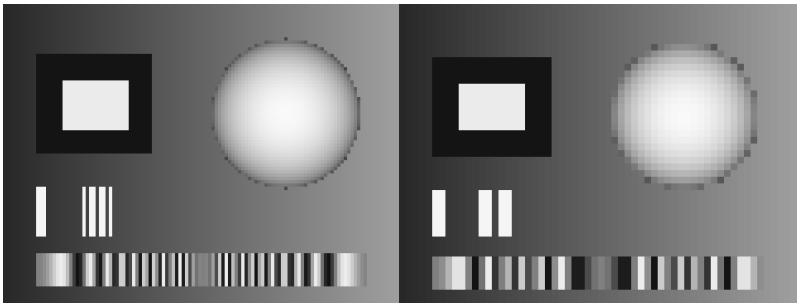
Now look at the rectangles and the bar groups: down to 4-level quantization they remain crisply outlined. The gray-value difference across the step edge (20 versus 235) is far larger than the step size, so no matter where quantization places its cuts it cannot erase that contrast; the same holds for the fine bright bars (245) against the background. **Edges and fine structures are insensitive to quantization; smooth gradient regions are the most sensitive.** This conclusion maps directly onto bit-depth selection in engineering: inspection-type tasks built around edge finding, locating, and presence/absence checks care about high-contrast structures, and 8 bits is almost always enough; only measurement-type and photometric tasks, which must read information out of gentle gray-value variations, truly justify paying for higher bit depth.

1.3 Sampling and a First Look at Aliasing

Quantization loses gray values; sampling loses spatial detail. The intuitive criterion is: **to resolve a periodic structure, at least 2 sampling points must land in each period** — one for the bright half and one for the dark half — or the structure can slip through the gaps in the grid. This is the intuitive version of the sampling theorem; the rigorous frequency-domain statement is deferred to Chapter 11. When a structure is finer than this limit, it does not simply vanish — it **masquerades as a coarse structure that does not exist**. This phenomenon is called **aliasing**.

The experiment undersamples the scene spatially: keep 1 pixel

out of every 4 (or every 8), with no prefiltering of any kind, then enlarge back to the original size by pixel replication for comparison. The equivalent resolutions are 120×90 and 60×45. The results are shown in Figure 1.3.



(a) 1/4 sampling (equivalent 120×90)

(b) 1/8 sampling (equivalent 60×45)

Figure 1.3: The spatial undersampling experiment (no prefiltering; pixel replication back to the original size). (a) 1/4 sampling: the 1 px and 2 px bar groups either disappear or merge into thick bars, and the dense stripes on the right of the chirp band fold back into sparse fake coarse stripes (aliasing); (b) 1/8 sampling: the whole image is severely pixelated, the bar groups degenerate into scattered patches, and the sphere's edge shows pronounced jagging.

In Figure 1.3a, the two kinds of fine structure meet different fates, both equally bad. The 1 px bar group has a period of only 2 px, far below what a 4 px sampling interval can resolve: where a sampling point lands on a bar, the bar "exists"; where it lands in a gap, the bar vanishes. The result is that the original six neat thin bars either evaporate into thin air or fuse into one thick bar — a reader using this as a resolution target would reach a completely wrong conclusion. The chirp band is more deceptive still: its true frequency rises monotonically to the right, but in the undersampled image the stripes that should be densest on the right have instead become a set of slowly undulating **coarse** stripes. This "fake low frequency" is the standard face of aliasing — high-frequency structure beyond the sampling limit folds back and impersonates low frequency. The danger is that the result looks perfectly normal: the image

is not blurred; it is simply **wrong**, and the original content cannot be recovered from the undersampled data after the fact. The 1/8 sampling of Figure 1.3b pushes the destruction to the extreme: the bar groups are all but wiped out, the whole image turns into a strong mosaic of pixelation, and even the sphere's outline becomes jagged.

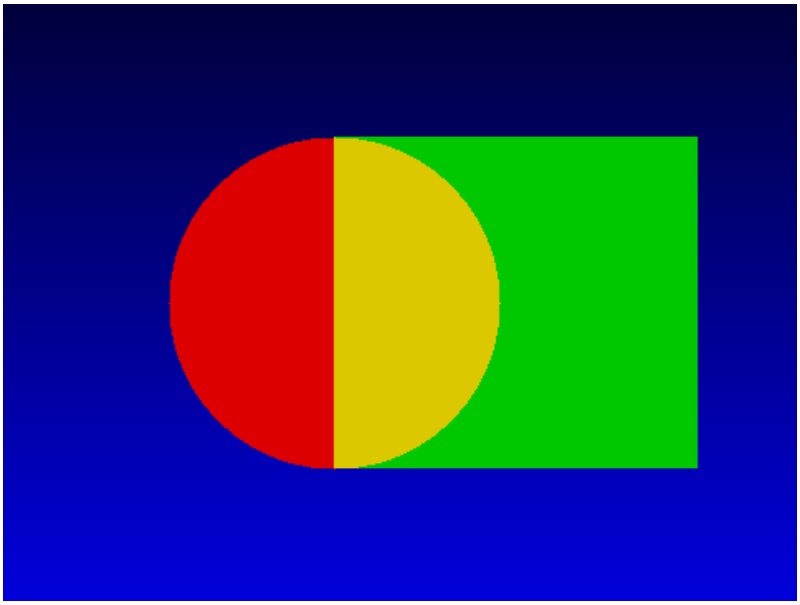
It is worth emphasizing that this experiment **deliberately omits prefiltering**. In a real camera, lens blur and the area integration of each photosite act as a natural low-pass prefilter that attenuates detail beyond the sampling limit before the photosites sample it, so aliasing is partly suppressed. Shrinking an existing digital image by "take every k-th pixel" decimation has no such protection — which is exactly why the engineering rule says "smooth before you downscale". The systematic treatment is given in Chapter 11.

The rule of thumb in industrial lens and camera selection that "**pixel resolution should be at least half the minimum defect size**" (i.e., the smallest defect spans 2 pixels) comes straight from the 2-samples-per-period criterion: only when a defect covers at least two pixels is it guaranteed to leave a detectable trace no matter where it falls on the grid. In practice the rule is often relaxed further to 3–5 pixels to leave signal-to-noise margin.

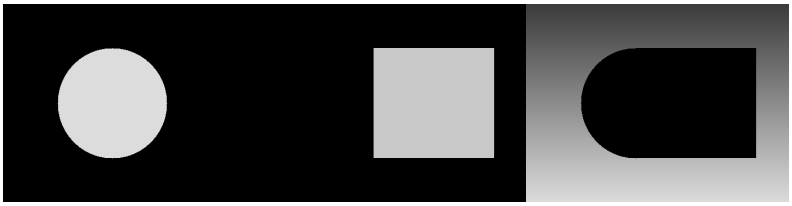
1.4 Color Images and Channels

Structurally, a color image contains nothing new: an RGB image is simply **three grayscale images of the same size** — the red, green, and blue **channels** — stacked together. The synthetic scene of Figure 1.4 is drawn channel by channel: the blue channel is a vertical gradient background running 60→220 from top to bottom, the green channel is a rectangle of brightness 200, and the red channel is a disk of brightness 220. Where the disk and the rectangle overlap, red and green are both bright, and additive mixing renders the region yellow; extract the three channels individually and each one is an ordinary grayscale image.

There are two schemes for laying the three channels out in memory. **Interleaved** storage places the three components of each pixel contiguously, pixel after pixel, so one row occupies 3 times the width in bytes; **planar** storage keeps each channel as one contiguous grayscale block, with the three blocks placed one after another. The interleaved format makes it convenient to grab "the complete color of one pixel" and is the default for camera output and most image libraries; the planar format is convenient for whole-channel operations.



(a) Color composite scene



(b) R channel (c) G channel (d) B channel

Figure 1.4: A color image and its channel decomposition. (a) A red disk and a green rectangle on a blue gradient background; the overlap appears yellow; (b) the R channel contains only the disk; (c) the G channel contains only the rectangle; (d) the B channel is the gradient background, with the regions occupied by the disk and the rectangle at 0 (black holes).

Interleaved storage carries one more detail you must watch: **the order of the three components**. In the SciVision SDK used throughout this book, the interleaved order of a 3-channel `SciImage` is **BGR** (byte 0 is blue) — the same convention as OpenCV's default, and the reverse of the literal order "RGB". This is stated nowhere in the SDK headers; we confirmed it empirically with a probe experiment. The first time this chapter's color scene was generated, the disk came out blue and the background red — the classic symptom of a wrong channel-order assumption. Getting the channel order backwards raises no error, causes no crash, and the image even looks "normal" — red and blue are merely swapped — which is fatal to any sorting algorithm that relies on color.

Finally, the choice of whether to use color at all. The information color carries is intuitive to the human eye, but to an algorithm it means 3 times the data volume and bandwidth. Most industrial tasks — edge finding, locating, dimensional measurement, defect inspection — depend on brightness structure rather than hue, and a grayscale camera with well-designed lighting is usually the optimum. The cases where color is genuinely irreplaceable are tasks in which the targets **differ only in color**: identifying resistor color bands, sorting colored cables, checking color deviation in printed matter. Even with a color camera, the first step of many algorithms is to pick out the single channel with the best contrast and use it as a grayscale image.

The probe method: construct an image in which only a single channel is nonzero (e.g., write only byte 0), save it, and open it with a tool whose channel order is known (OpenCV or any image viewer); the displayed color tells you whether the undocumented SDK uses RGB or BGR. One experiment beats any amount of guessing.

1.5 Image Data Structures and SciVision

All images in this chapter's experiments are constructed via `SciImage` on top of a memory buffer. The construction statements for the grayscale and color images are as follows (excerpted from `code/digital_images/main.cpp`):

```
SciImage img(copy.data(), W, H, W, 1, SCI_IMAGE_8U);        // grayscale: step = W
SciImage color(inter.data(), W * 3, H, W, 3, SCI_IMAGE_8U); // color: step = W*3
```

The full constructor signature is `SciImage(unsigned char*`

`data, int step, int rows, int cols, int channel, SciImageType depth)`. The six parameters, one by one:

- `data`: the address of the pixel buffer. `SciImage` does not copy the data; the buffer's lifetime must cover the entire period the image object is in use.
- `step`: **the number of bytes per row** (the row stride). It is not "the number of pixels per row" — for a 1-channel 8-bit image it happens to equal the width `W`, but for a 3-channel image it is `W * 3`; and if each row of the buffer ends with alignment padding bytes, `step` must include the padding. When passing images between libraries goes wrong, nine times out of ten the bug is in step.
- `rows`, `cols`: the number of rows (height `H`) and columns (width `W`). Note the constructor order: step first, then rows, then columns.
- `channel`: the number of channels — 1 for grayscale, 3 for color; with 3 channels the data is interleaved in BGR order (the empirically verified finding of the previous section), and the writing code is:

```
for (int i = 0; i < W * H; ++i) {       // BGR interleaved (SDK order, verified by probe)
    inter[i * 3 + 0] = cb[i];
    inter[i * 3 + 1] = cg[i];
    inter[i * 3 + 2] = cr[i];
}
```

- `depth`: the pixel-type enumeration (defined in `SciDataDef.h`). `SCI_IMAGE_8U = 0` means 8-bit unsigned integers per channel — the 8-bit images this chapter has been discussing throughout; higher-bit-depth data corresponds to other values of the same enumeration.

The other core structure paired with `SciImage` is `SciROI` — the **region of interest (ROI)**. In production-line images, usually only a small patch actually needs processing; restricting the algorithm to an ROI both saves time and avoids irrelevant interference, and the algorithm interfaces in nearly every later chapter of this book take an ROI parameter. How to generate the ROI dynamically from the workpiece's actual position (position correction) is covered in detail in Chapter 19.

The complete runnable project that generates every experimental image in this chapter is located at `code/digital_images/`; readers can modify the quantization levels and sampling intervals to reproduce the results themselves.

Industry Case: 8-bit or 12-bit?

A backlight-panel brightness-uniformity measurement project initially chose an 8-bit camera. In the panel's dark corner regions, all the gray values were squeezed into a dozen or so gray levels; the jump between adjacent levels alone consumed several percent of the measurement range, the resolution of the brightness computation fell far short of the acceptance requirement, and repeated measurements would not converge. After switching to a 12-bit camera, the same dark regions had 16 times as many gray levels, the measurement resolution immediately met the specification, and the problem was solved. But the cost was just as real: 12-bit pixels take 1.5 times the storage and transmission bandwidth, the camera's frame rate dropped accordingly, and the image buffers along the entire pipeline had to be re-budgeted. The post-mortem conclusion: blindly choosing high bit depth for inspection-type tasks is pure waste — bit depth should be chosen according to the task's need for **gray-value resolution**, not on the principle of "the higher the better".

1.6 Summary

- **A digital image = sampling + quantization**: sampling determines spatial resolution, quantization determines the number of gray levels (bit depth); the two lose information independently, and neither loss is recoverable.
- **Quantization hurts smooth gradient regions first** (banding, concentric rings) while leaving step edges and high-contrast fine structures almost untouched — 8 bits is usually enough for inspection-type tasks, and only measurement-type and photometric tasks that use gray values as physical quantities need 10/12 bits.

- **Fine structures need at least 2 sampling points per period**; when a signal is undersampled, high frequencies beyond the limit do not disappear — they alias into convincing fake low-frequency structures that cannot be repaired afterwards. Smooth before downscaling, and when selecting hardware make sure the smallest defect covers at least 2 pixels.
- **A color image is three grayscale images**; with interleaved storage, always verify the channel order: the 3-channel `SciImage` in SciVision was empirically found to be BGR, with no mention in the headers — when integrating any SDK, one probe experiment beats guessing.
- **`SciImage`'s `step` is bytes per row, not pixels per row**; multiple channels and row alignment both pull it away from the width value, making it the most common failure point when passing images across libraries.

A systematic treatment of sampling, quantization, and color spaces is given in (Gonzalez and Woods 2018), whose coverage of colorimetry and the various color models is especially complete; a modern perspective on image formation and digitization can be found in (Szeliski 2022). The rigorous origin of the sampling theorem's "at least 2 samples per period" is Shannon's foundational paper (Shannon 1949). For a more engineering-oriented treatment of image acquisition, sensor characteristics, and the digitization chain of industrial cameras, see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

2 Mathematical Preliminaries

The daily life of industrial vision engineering looks like tuning parameters, configuring lighting, and writing pipelines, but underneath it the same cast of mathematical characters is always on stage: "straightening up" a part relies on matrix multiplication, turning a string of edge points into a datum line relies on least squares, deciding whether a gray value counts as anomalous relies on probability distributions, and pushing localization accuracy from whole pixels to subpixels relies on interpolation. In university curricula this material is scattered across three courses — linear algebra, probability and statistics, and numerical analysis. This chapter gathers **the small slice that genuinely keeps reappearing on the production line**, selected by one criterion: which later chapter will need it. We aim not for completeness, but for every formula to have a definite destination. The table below is the learning map for this chapter:

Section	Content	Later chapters it supports
Section 2.1	Vectors, transformation matrices, homogeneous coordinates, eigenvectors	Chapter 10, Chapter 5, Chapter 14
Section 2.2	Normal equations, weighted least squares, numerical stability	Chapter 14, Chapter 5

Section	Content	Later chapters it supports
Section 2.3	Expectation and variance, the Gaussian distribution, histograms, the central limit theorem	Chapter 6, Chapter 7, Chapter 26
Section 2.4	Bilinear interpolation, three-point parabolic interpolation	Chapter 10, Chapter 20, Chapter 14

A note up front: this is a theory chapter, with no standalone companion code project; every conclusion derived here will be cashed in repeatedly in the real experiments of later chapters — for example, Chapter 6 uses Gaussian-noise experiments with $\sigma = 18$ to confirm that "averaging cancels independent errors", and Chapter 14 uses an inlier-RMSE comparison of 0.723 px versus 0.257 px to demonstrate the squared penalty's sensitivity to outliers. Readers can treat this chapter as a "memo with proofs": whenever a formula in a later chapter leaves you unsure of its origin, flip back here.

2.1 Vectors, Matrices, and Geometric Transformations

Throughout the book, bold lowercase letters denote **vectors**, always column vectors: a continuous coordinate point in the image plane is written $\mathbf{x} = (x, y)^\top$, consistent with the Notation chapter; in a discrete image $f[n, m]$, n is the row index (downward) and m is the column index (rightward). **Matrices** are denoted by uppercase letters. A 2×2 matrix A acting on a point ($\mathbf{x}' = A\mathbf{x}$) is a linear transformation: rotation, scaling, shear — each is some specific A.

The three 2D transformations industrial vision uses most form a family ordered from strict to permissive. A **rigid transformation** contains only rotation and translation:

$$\mathbf{x}' = R\,\mathbf{x} + \mathbf{t}, \qquad R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix},$$

with 3 degrees of freedom (the angle θ plus the two components of the translation $\mathbf{t}$), and preserves all lengths and angles — exactly the mathematical statement of "the part shifted and rotated on the stage, but the part itself did not change", and the standard model for position correction. A **similarity transformation** adds a global scale factor s on top of the rigid one (that is, $\mathbf{x}' = sR\,\mathbf{x} + \mathbf{t}$, 4 degrees of freedom): lengths are no longer preserved, but angles and shape still are — suitable when the working distance changes and the image grows or shrinks. An **affine transformation** opens this up further to an arbitrary invertible 2×2 matrix plus translation (6 degrees of freedom), allowing independent scaling along the two axes and shear; straight lines remain straight and parallel lines remain parallel, but angles are no longer preserved. Chapter 10 will put this family to work for image resampling and position correction.

Note that all three expressions above are written as "matrix multiplication **plus** translation" — the translation cannot be folded into the 2×2 matrix, because a linear transformation must map the origin to the origin. **Homogeneous coordinates** solve this with a simple dimension lift: append a 1 to every point, writing $\tilde{\mathbf{x}} = (x, y, 1)^\top$, and the entire affine transformation becomes a single 3×3 matrix multiplication:

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

Finally, the geometric intuition of the **eigenvector**, which will take center stage in the closed-form line-fitting solution of Chapter 14. For a set of points $\{\mathbf{p}_i\}_{i=1}^N$, write the centroid as $\bar{\mathbf{p}} = \frac{1}{N}\sum_i \mathbf{p}_i$ and define the **scatter matrix**

The engineering value of homogeneous coordinates lies in **chaining**: image coordinates → distortion correction → position correction → physical coordinates — each step is a 3×3 matrix, and the whole chain can be pre-multiplied into **one** matrix, so at run time each point costs a single multiplication. If the last row is not restricted to $(0, 0, 1)$, we obtain the perspective transformation (homography) — Chapter 5 cannot do without it when describing how a planar calibration target is imaged.

$$S = \sum_{i=1}^{N} (\mathbf{p}_i - \bar{\mathbf{p}})(\mathbf{p}_i - \bar{\mathbf{p}})^\top,$$

a 2×2 symmetric matrix. Its two eigenvectors are mutually perpendicular, and the one whose **eigenvalue** is larger points in the direction along which the point set is **most spread out** — feed in a string of edge points distributed along some edge, and the principal eigenvector hands you the edge's direction. This is no coincidence: one can prove that the line minimizing the sum of squared perpendicular distances to the points passes exactly through the centroid, with its direction given by the principal eigenvector of S. Compute one centroid, solve one 2×2 eigenproblem, and line fitting has a closed-form solution — this is the entire content of the phrase "the solution is closed-form" in Chapter 14.

2.2 Least Squares and Robustness

Least squares is the engine of measurement-class algorithms, and the simplest case is worth deriving in full. Given N points (x_i, y_i), we want to fit the line $y = kx + b$, defining "fits well" as minimizing the sum of squared vertical residuals:

$$E(k, b) = \sum_{i=1}^{N} (kx_i + b - y_i)^2.$$

E is a smooth convex function of k, b, so at the minimum the partial derivatives vanish:

$$\frac{\partial E}{\partial k} = 2 \sum_i x_i (kx_i + b - y_i) = 0, \qquad \frac{\partial E}{\partial b} = 2 \sum_i (kx_i + b - y_i) = 0.$$

Rearranged into a linear system in (k, b), these are the **normal equations**:

$$\begin{pmatrix} \sum_i x_i^2 & \sum_i x_i \\ \sum_i x_i & N \end{pmatrix} \begin{pmatrix} k \\ b \end{pmatrix} = \begin{pmatrix} \sum_i x_i y_i \\ \sum_i y_i \end{pmatrix}.$$

Two rows, two columns — one solve yields k, b; and the second row throws in a property worth remembering: $b = \bar{y} - k\bar{x}$, that is, **the least-squares line always passes through the centroid of the point set**. The general case is completely isomorphic: write the model as $A\theta \approx \mathbf{y}$ (each row of A is one observation, θ the parameters to be estimated), and the normal equations are $A^\top A\,\theta = A^\top \mathbf{y}$. Whether it is circle fitting, homography estimation for a calibration target, or solving the hand–eye relation, most problems eventually land in this form.

Two common extensions. First, **weighted least squares**: if the observations differ in trustworthiness, give each term a weight $w_i \geq 0$ and minimize $\sum_i w_i r_i^2$; the normal equations become $A^\top W A\,\theta = A^\top W \mathbf{y}$ (W the diagonal weight matrix) — the derivation is word-for-word the same as above, only with an extra w_i inside every sum. Second, replace the "vertical residual" with the **perpendicular distance** from point to line: the optimum is then no longer given by the normal equations, but is exactly the closed form of the previous section — through the centroid, direction along the principal eigenvector of the scatter matrix; it favors no coordinate axis and is the right way to do geometric fitting.

The Achilles' heel of least squares hides in that same square: a single spurious point with a 12 px residual contributes nearly 600 times as much to the objective as a normal point with a 0.5 px residual, and the fit can be hijacked by a tiny minority of outliers — the controlled experiment in Chapter 14 puts striking numbers on this (inlier RMSE 0.723 px versus 0.257 px). The cure is to replace the squared penalty with one more forgiving of large residuals (such as Huber weighting (Huber 1964), solved iteratively via weighted least squares) or with random sample consensus, RANSAC (Fischler and Bolles 1981); both are left for that chapter to develop alongside the experiments. Here it suffices to remember the root cause: **squaring amplifies the voice of large residuals**.

The **condition number** measures how much a solution amplifies perturbations of the input, and the condition number of $A^\top A$ is the **square** of that of A. If you fit directly in pixel coordinates (easily in the thousands), $\sum x_i^2$ and N differ by six orders of magnitude, the matrix is nearly ill-conditioned, and floating-point errors are amplified sharply; circle fitting contains an $x^2 + y^2$ term, making things even worse. The engineering rule: **first subtract the centroid to center the data** (and, if necessary, divide by a scale to normalize), solve in small coordinates, then transform the result back — a few lines of code buy back several significant digits.

2.3 Probability and Statistics Essentials

Noise cannot be predicted pixel by pixel, but it can be characterized statistically. Treat a single gray-value observation as a **random variable** X; its two most important summaries are the **expectation** $\mu = \mathrm{E}[X]$ (what repeated observations average toward) and the **variance** $\sigma^2 = \mathrm{E}[(X-\mu)^2]$ (how spread out values are around the expectation). The square root of the variance, σ, is the standard deviation; it carries the same units as the gray values, and in practice serves directly as the "noise amplitude". Together, the linearity of expectation and the additivity of variance for independent observations immediately yield a law used every day on the production line: averaging N independent, identically distributed observations gives a mean whose variance is $1/N$ of a single observation's, i.e. the standard deviation shrinks to $\sigma/\sqrt{N}$ — multi-frame averaging, the neighborhood smoothing of Chapter 6, and the projection averaging along search lines in Chapter 14 are all incarnations of this $\sqrt{N}$ law.

The most important distribution in machine vision is the **Gaussian distribution** $\mathcal{N}(\mu, \sigma^2)$, with probability density

$$p(x) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

Its values are highly concentrated: the probabilities of falling within $\mu \pm \sigma$, $\mu \pm 2\sigma$, and $\mu \pm 3\sigma$ are about 68.3%, 95.4%, and 99.7% respectively — the last is the famous **3 rule**. It gives threshold setting a direct vocabulary: if the gray values of a good region follow $\mathcal{N}(\mu, \sigma^2)$, then flagging "outside $\mu \pm 3\sigma$" as anomalous yields a per-pixel false-alarm rate of only about 0.3%. But beware the effect of scale: in a five-megapixel image, 0.3% means over ten thousand pixels cross the line out of sheer luck, which is why defect-detection practice often loosens the gate to 4σ–6σ, or follows the 3 test with a connected-area filter — Chapter 26 will return to this accounting.

Where does the distribution of a real image come from? Just count: tally how many pixels take each gray level and you get the **histogram**; divide by the total pixel count and it becomes

Estimating σ itself also needs protection against outliers: the sample standard deviation contains a squared term, and a few bad points can inflate it. The robust alternative is the **median absolute deviation (MAD)**: $\hat{\sigma} = 1.4826 \times \mathrm{median}_i\,(|x_i - \mathrm{median}(x)|)$, where the factor 1.4826 makes it agree with the standard deviation on Gaussian data. The median naturally ignores a small number of extreme values, which is why MAD is a standard fixture of threshold statistics on the line.

the empirical estimate of the gray-value distribution — expectation and variance can both be computed straight from it. An image of a dark part on a bright background shows a bimodal histogram, and the valley between the two peaks is a natural segmentation threshold; Otsu's method in Chapter 7 treats the histogram precisely as a probability distribution and searches for the threshold maximizing the between-class variance — the expectation and variance of this section are its entire raw material.

Finally, an answer to a question that has hung over two chapters: why model sensor noise as Gaussian? The answer is the **central limit theorem**: the sum of a large number of mutually independent, individually tiny random perturbations tends toward a Gaussian distribution, regardless of what each perturbation's own distribution is. Pixel readout perturbations are exactly this situation — thermal fluctuations of dark current, readout-circuit noise, and electronic noise from gain amplification stack layer upon layer, each small and independent, so the total effect is approximately Gaussian. This is the justification for Chapter 6 running all of its controlled experiments with Gaussian noise at $\sigma = 18$; it also explains that chapter's counterexample — salt-and-pepper noise comes from **single, large-amplitude** events such as dead pixels and transmission errors, violating the "many tiny independent perturbations" premise, so it is not Gaussian, and methods built for Gaussian noise should not be used against it.

2.4 Interpolation

An image has values only at integer grid points, yet algorithms keep demanding values at non-integer positions: rotate an image, and a target pixel mapped back into the source almost never lands on a grid point (Chapter 10); take a gray profile along a search line in an arbitrary direction, and the sample points fall between grid points too (Chapter 20). **Interpolation** is responsible for answering "what lies between the grid points".

The cheapest answer is **nearest-neighbor interpolation**:

take the value of the closest grid point. It creates no new gray values and is the fastest, but positionally it amounts to rounding by up to 0.5 pixel; edges come out jagged after geometric transformations, so it is essentially never used for measurement. The standard answer is **bilinear interpolation**: let the sample point fall inside the unit square bounded by rows n, $n+1$ and columns m, $m+1$, with fractional offsets $\beta, \alpha \in [0, 1)$ in the row and column directions; then

$$f(n+\beta,\, m+\alpha) = (1-\alpha)(1-\beta)\, f[n,m] + \alpha(1-\beta)\, f[n,m+1] + (1-\alpha)\beta\, f[n+1,m] + \alpha\beta\, f[n+1,m+1].$$

The four weights are exactly the four **diagonally opposite areas** into which the sample point divides the unit square: the closer the point is to a grid point, the larger that grid point's say; the four weights always sum to 1, so a flat region interpolates to a flat region. Bilinear output is continuous and cheap to compute, making it the default choice for industrial geometric transformations; the higher-order bicubic interpolation trades a 4×4 neighborhood for a smoother result, and Chapter 10 will compare them on real measurements.

The other kind of interpolation serves not resampling but **finding the precise location of an extremum** — the mathematical core of subpixel localization. A discrete extremum of a one-dimensional profile is accurate only to the whole pixel: place the extremum at local coordinate $u = 0$ with value g_0, and its left and right neighbors at $u = \mp 1$ with values g_{-1}, g_{+1}. The true continuous extremum need not sit at $u = 0$, but three points suffice to determine a parabola $g(u) = au^2 + bu + c$, whose vertex approximates the true extremum. Substituting the three samples:

$$g(0) = c = g_0, \qquad g(\pm 1) = a \pm b + c = g_{\pm 1},$$

and adding and subtracting the two equations gives immediately

$$a = \frac{g_{+1} + g_{-1} - 2g_0}{2}, \qquad b = \frac{g_{+1} - g_{-1}}{2}.$$

The parabola's vertex lies where the derivative vanishes, at $u^* = -b/(2a)$; substituting a, b:

$$\delta = u^* = \frac{g_{-1} - g_{+1}}{2\,(g_{-1} - 2g_0 + g_{+1})}.$$

This is precisely the subpixel formula of step 4 in the edge-localization procedure of Chapter 14. Two sanity checks deepen one's trust in it. First, if $g_{-1} = g_{+1}$ (left–right symmetric), then $\delta = 0$ and the extremum sits exactly on the whole pixel, as intuition demands. Second, as long as g_0 is a strict discrete maximum, the denominator satisfies $g_{-1} - 2g_0 + g_{+1} < 0$ without exception and $|\delta| \le 1/2$ — the vertex never escapes past the midpoint to a neighboring pixel, so the formula is numerically safe. Three samples and one division push localization accuracy from the 1-pixel scale to the 0.1-pixel scale — that is the entire cost behind "subpixel accuracy is almost free".

Industry Case: Three Small Things That Make Measurements Right

A vision engineer at a precision structural-parts plant once summed up three "small math" lessons at a postmortem meeting. First: an early roundness-inspection program fitted circles directly in full-frame pixel coordinates — coordinate values in the thousands, and with the $x^2 + y^2$ term the normal equations' condition number was enormous; measuring the same part ten times in a row, the fitted center drifted by 0.3 px. Subtracting the centroid before fitting and adding it back afterwards dropped the drift to 0.02 px on the spot. Second: when cleaning edge points with "mean ± 3 rejection", the was computed with the bad points included — the bad points first inflated , then sailed through the loosened gate en masse; only after switching to a robust estimated by MAD, or to iterative re-estimation after rejection, did the gate actually hold. Third: a line-width measurement station's repeatability stubbornly hovered around 0.1 px off target; lighting and vibration were investigated to no avail, until it emerged that the profile sampling used nearest-neighbor interpolation — switching to bilinear brought the repeatability within spec immediately. Not one of the three involved any deep algorithm; all of it is in the few pages of this chapter.

2.5 Summary

- **Homogeneous coordinates unify "matrix multiplication plus translation" into a single 3×3 multiplication**, and a chained sequence of transformations can be pre-multiplied into one matrix; rigid (3 degrees of freedom), similarity (4), and affine (6) form the 2D transformation family from strict to permissive, preserving lengths, shape, and parallelism respectively.
- **The normal equations come from "set the partial derivatives of the objective to zero"**: the general form is $A^\top A\,\theta = A^\top \mathbf{y}$, and the weighted version just multiplies each sum by w_i; the closed-form solution for fitting a line by perpendicular distance is "through the centroid, direction along the principal eigenvector of the scatter matrix".
- **Numerical stability is part of measurement accuracy**: $A^\top A$ squares the condition number, so with large coordinates always center the data before fitting; the squared penalty amplifies the voice of outliers, and robust fitting (Huber, RANSAC) takes over in Chapter 14.
- **The Gaussian distribution plus the 3 rule is the vocabulary of threshold setting**, and the central limit theorem explains why sensor noise is nearly Gaussian; the histogram is the empirical estimate of the distribution, and averaging makes noise converge as $\sigma/\sqrt{N}$ — smoothing, projection, and multi-frame averaging all share this one law.
- **Bilinear interpolation weights the four neighbors by diagonally opposite areas**; three-point parabolic interpolation gives the vertex offset $\delta = (g_{-1} - g_{+1})/(2(g_{-1} - 2g_0 + g_{+1}))$, with $|\delta| \le 1/2$ always — it is the source of every subpixel formula in this book.

A systematic treatment of the probability, statistics, and Gaussian-model thread is given in (Bishop 2006); the normal equations, condition number, and numerical stability of least squares are classic numerical-linear-algebra material, for which the authoritative reference is (Golub and Van Loan 2013); a quick reference on geometric transformations, linear algebra,

and optimization in a vision context can be found in the appendices of (Szeliski 2022). This chapter's material is developed more completely, and closer to industrial measurement, in the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018), particularly the parts on geometric transformations and on least-squares fitting with its uncertainty analysis, which make good further reading.

Part II

Imaging and Calibration

This part explains how to acquire high-quality images: camera and lens selection and imaging models, light source and illumination design, and the principles and engineering practice of camera calibration.

3 Cameras and Lenses

However ingenious the algorithms in a machine vision system may be, they only ever process the image the camera hands over. At the very top of the algorithmic chain sits the optics: the moment light passes through the lens and lands on the sensor, the upper bound on image quality is already fixed — insufficient contrast, smeared detail, motion blur, perspective distortion: none of these defects can be truly repaired by any downstream algorithm. Spending money and thought on choosing the right camera and lens is usually a far better deal than "enhancing" the image in software after the fact.

This chapter does three things. First, we establish the pinhole imaging model and derive the selection calculations that link focal length, working distance, and field of view. Next, we discuss the key engineering concepts on the sensor and lens side — pixels, shutters, depth of field, and telecentricity. Finally, we answer a question that gets asked on the production line every single day: **can "sharp" be turned into a number?** We will present two families of focus measures, compare their merits with a set of real experiments, and explain how autofocus "climbs the hill" along the resulting curve.

3.1 The Pinhole Model and Imaging Geometry

The simplest and most useful camera model is the **pinhole model**: all rays pass through an ideal small aperture and form an inverted image on an image plane at distance f behind it. Let a point in space have coordinates (X, Y, Z) in the camera frame (Z being the distance along the optical axis); its image-point coordinates are

$$x = f\,\frac{X}{Z}, \qquad y = f\,\frac{Y}{Z}.$$

This is **perspective projection**, and its heart is that division by Z: an object of the same size images smaller the farther it is from the camera (the larger Z is) — the everyday experience of "near things look big, far things look small" compressed into a single division. The parameter f is the **focal length**, the scaling knob of the imaging geometry: the longer f, the larger the image of a given object and the narrower the field of view.

The quantities used most often in lens selection are connected through this model. Let the **working distance (WD)** be the distance from the lens to the object, the **field of view (FOV)** be the width of the object plane that the image just covers, and h be the sensor width; then the **magnification** is

$$m = \frac{h}{\mathrm{FOV}}, \qquad f = \frac{m \cdot \mathrm{WD}}{1+m}.$$

The second equation follows from the thin-lens formula; when $\mathrm{WD} \gg f$ it degenerates into the more familiar rough form $f \approx m \cdot \mathrm{WD}$.

Real lenses also exhibit distortion: straight lines bend into arcs on the image plane, increasingly so toward the edge of the field. Chapter 5 will add distortion terms and the intrinsic matrix on top of the pinhole model to obtain a complete model fit for precision measurement; in this chapter the ideal pinhole is enough to build selection intuition.

Let us work a concrete example. An inspection station uses a 1/1.8 sensor with 2.4 m pixels and a resolution of 3072×2048, giving a sensor width of $h = 3072 \times 2.4\ \mu\mathrm{m} \approx 7.4$ mm; the mechanics fix the working distance at 300 mm, and a field of view 50 mm wide must be covered. Thus $m = 7.4/50 \approx 0.148$, and substituting gives $f \approx 0.148 \times 300/1.148 \approx 38.7$ mm. No off-the-shelf fixed-focal-length lens comes in a 38.7 mm step, so we **round down** to the nearest 35 mm: the actual magnification becomes $m' = f/(\mathrm{WD} - f) = 35/265 \approx 0.132$, and the actual field of view about $7.4/0.132 \approx 56$ mm — slightly wider than required, which leaves margin for mounting and alignment; choosing 50 mm instead would shrink the field to about 37 mm, simply not enough. Finally, check the spatial resolution: 56 mm spread over 3072 pixels gives about 18 m per pixel. Whether that number suffices depends on the rule of thumb in the next section.

3.2 Sensors and Pixels

A **pixel** is a photosensitive cell on the sensor — in essence a "well" that collects photogenerated electrons. The maximum number of electrons one well can hold is called the **full well capacity**; the ratio of full well capacity to readout noise determines the **dynamic range** — the span between the brightest and darkest values that can be distinguished within a single image. Chapter 1 discussed bit depth: whether 8-bit or 12-bit quantization is meaningful comes down, in the end, to whether the sensor's dynamic range can support that many gray levels — otherwise the extra bits merely quantize the noise more finely.

This leads to a pair of frequently overlooked trade-offs: **for a sensor of the same size, higher resolution means smaller pixels**. As pixels shrink, the full well capacity scales down with the photosensitive area, the signal-to-noise ratio and dynamic range degrade in step, and brighter illumination or longer exposure is needed to feed them. "The higher the resolution, the better" is the most common fallacy in camera selection — resolution should be calculated from the requirement, not maxed out to the budget.

Another question that must be settled before the purchase order goes out is the shutter type. A **global shutter** starts and ends the exposure of all pixels at the same instant, yielding a true "single moment in time"; a **rolling shutter** exposes row by row, so the top and bottom of one frame are separated by a full readout period. For a stationary workpiece the two are indistinguishable, but the moment the workpiece or the camera moves, a rolling shutter turns rectangles into parallelograms and circles into eggs — a geometric distortion caused by the row-by-row time offset that no calibration can remove after the fact. **The conclusion is hard: workpieces in motion (fly-by capture on a conveyor, snapshots on a moving stage) require a global shutter.** Rolling-shutter sensors are cheaper and often lower in noise, and are suitable only for stationary-capture scenarios.

How is the requirement calculated? Rule of thumb: **the smallest defect or smallest feature must cover at least 2 pixels** — exactly the engineering version of the sampling theorem from Chapter 1 (in practice, 3–4 pixels are often used for margin). The example in the previous section gave 18 m per pixel, meaning the smallest reliably detectable defect is about 36–70 m; if the process requires detecting 20 m defects, the field of view must be shrunk or a higher-resolution solution adopted.

3.3 Depth of Field and Telecentric Lenses

In the pinhole model everything is in focus everywhere; a real lens has only one plane of focus. When an object departs from that plane, each object point spreads into a small blur spot on the image plane; as long as the spot diameter does not exceed the **circle of confusion** c (in engineering practice typically 1–2 pixels), both humans and algorithms still judge it "sharp". The front-to-back range over which sharpness is maintained is called the **depth of field (DOF)**, approximately

$$\mathrm{DOF} \;\approx\; \frac{2\,c\,N\,(m+1)}{m^2},$$

where N is the f-number. Read this formula qualitatively: **stopping down the aperture (increasing N) increases the depth of field** — at the cost of light throughput falling as N^2, and beyond a certain point diffraction makes the whole field soft instead; **the magnification m sits in the denominator squared**, so the depth of field collapses extremely fast at high magnification — microscopic observation has a sharp layer as thin as paper. When the depth of field is insufficient, first consider stopping down and adding illumination, then reducing the magnification, and only as a last resort flattening the workpiece mechanically.

Ordinary lenses also have an inherent problem unrelated to focus: perspective error. From $x = fX/Z$, the magnification varies with Z: $\Delta m/m \approx \Delta Z\,/\,Z$. Returning to the example of the previous section: with a working distance of 300 mm, a workpiece surface undulation or fixture repeatability of ± 1 mm changes the magnification by $\pm 0.33\%$, introducing a reading drift of ± 0.17 mm over a 50 mm measured dimension — catastrophic for dimensional measurement with tolerances of a few micrometers. And this is not a focus problem; stopping down will not save you.

The **telecentric lens** exists precisely for this: the aperture stop is placed at the image-side focal plane, so that all object-side chief rays are parallel to the optical axis. Within the telecentric range, therefore, **the magnification does not vary with**

object distance — the workpiece can wobble, be thicker, sit tilted, and the image size does not budge; perspective error is eliminated by the optical structure itself. The price is size, weight, and cost. When is a telecentric lens mandatory? High-precision dimensional measurement (tolerances down at the micrometer level), measured objects with thickness or steps (an ordinary lens images the upper and lower surfaces at different magnifications), and situations where the Z position cannot be tightly controlled.

The hard constraint of telecentric lenses: because the chief rays are parallel to the optical axis, **the front group's aperture must be no smaller than the measured object** — measuring a 50 mm workpiece requires a telecentric lens with an aperture exceeding 50 mm. This is the fundamental reason telecentric lenses are large and expensive, and why they are normally used only at measurement stations, not for large-field locating.

3.4 Quantitative Evaluation of Sharpness

In the lab, "is the image sharp?" is judged by eye; on the production line, it must be judged by a number: autofocus needs an objective function to search over; equipment alignment and acceptance need a metric that can be written into the process document; comparing old and new lenses, or lenses from different batches, needs one common yardstick. A function that maps an image to a single scalar for this purpose is called a **focus measure**; a good one takes a unique peak at best focus and decreases monotonically with defocus.

The most direct idea comes from observation: when an image blurs, what is lost is high-frequency detail — edges become gentle. The **gradient-energy method** (the Tenengrad idea) directly measures how "steep" the edges are:

$$S_{\text{grad}} = \frac{1}{NM} \sum_{n,m} \Big(G_x[n,m]^2 + G_y[n,m]^2 \Big),$$

where G_x and G_y are the horizontal and vertical gradients (e.g., Sobel operator responses). The sharper the image, the larger the gradient magnitudes and the higher the score. The other family is the **gray-level variance method**, which measures how far the whole image's gray values depart from their mean:

$$S_{\text{var}} = \frac{1}{NM} \sum_{n,m} \left(g[n,m] - \bar{g} \right)^2, \qquad \bar{g} = \frac{1}{NM} \sum_{n,m} g[n,m].$$

Blur is a form of averaging: it squeezes gray values toward the mean, and the variance drops accordingly — it is the fastest to compute, but what it measures is **global contrast**, not detail. Which of the two is better? Let the experiments speak.

We synthesize a 480×360 test scene (Figure 3.1): the top row holds four groups of vertical stripes with periods of 2, 4, 6, and 8 px, the equivalent of a simple resolution chart; the middle row holds a 4 px checkerboard and a block of random speckle simulating text-like texture; the bottom row holds a dark rectangle providing step edges and a diagonal black-and-white edge; in addition, two bright lines 1 px wide run across the frame. From the finest 1 px structures to the coarsest large light-and-dark blocks, every spatial scale is represented.

We then simulate progressive defocus with Gaussian blur (a real defocus spot is approximately a uniform disk of confusion; the Gaussian is its smooth first-order approximation — see Chapter 6 for the filter itself), with σ taking the seven levels 0, 0.8, 1.6, 2.4, 3.2, 4.8, 6.4, and the kernel size chosen as the odd number near $K \approx 6\sigma$. Figure 3.2 shows two of the levels: at $\sigma = 2.4$, the fine stripe groups with periods of 2 px and 4 px have completely melted into uniform gray blocks — that detail is gone for good — while the coarse stripes, the rectangle, and the diagonal edge survive; at $\sigma = 6.4$, all stripes and the checkerboard are wiped flat, and only large patches of light and dark remain.

For each blur level we compute both scores and normalize them to 1 at $\sigma = 0$; the results are given in Table 3.1 and Figure 3.3.

Table 3.1: Normalized scores of the two focus measures versus blur level (raw scores at $\sigma = 0$: gradient method 66.4361, variance method 33.2610)

Blur σ	Gradient (normalized)	Variance (normalized)
0.0	1.0000	1.0000

Blur σ	Gradient (normalized)	Variance (normalized)
0.8	0.7330	0.6693
1.6	0.3441	0.5125
2.4	0.1850	0.4716
3.2	0.1356	0.4552
4.8	0.1145	0.4302
6.4	0.1053	0.4068

Both curves decrease strictly monotonically — either is "usable" as an autofocus objective — but the difference in quality is dramatic. **The gradient method discriminates far better than the variance method**: from $\sigma = 1.6$ to 6.4, the gradient score drops from 0.344 all the way to 0.105 (a span of more than 3×), while the variance score eases only from 0.513 to 0.407, hitting a plateau early. The reason goes back to the nature of the two formulas: what blur erases is high-frequency detail, while the large light-and-dark structures — the dark rectangle, the diagonal edge — retain their global contrast at any blur level. The variance method looks only at contrast, so even when the image has blurred into nothing but color patches it still awards forty percent of the score; the gradient method watches the steepness of edges and reacts far more sensitively to the loss of detail. A focus measure with poor discrimination means a focus curve with a flat peak region and a nearly flat far-defocus region — the search algorithm can neither judge direction nor decide convergence.

Given a monotonic focus curve, **autofocus is hill climbing on that curve**: move the focus motor or the Z axis, capture and score an image at each position, keep going in the direction of rising score, and stop once past the peak. The practical search strategy is **coarse-to-fine** in two stages: first sweep the whole travel with a large step to bracket the peak within an interval (also steering clear of noise artifacts on the far-defocus plateau); then scan that interval with a small step and fit a parabola to the few samples near the top, interpolating the best focus position to within a fraction of the step. The better the focus measure's discrimination, the more reliable the direction decisions in the coarse sweep, and the narrower the fine-scan interval can be drawn.

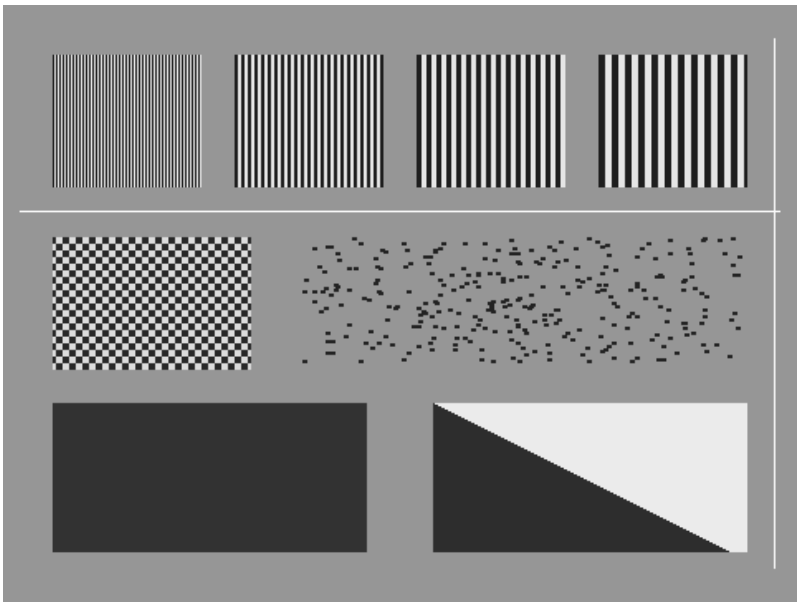
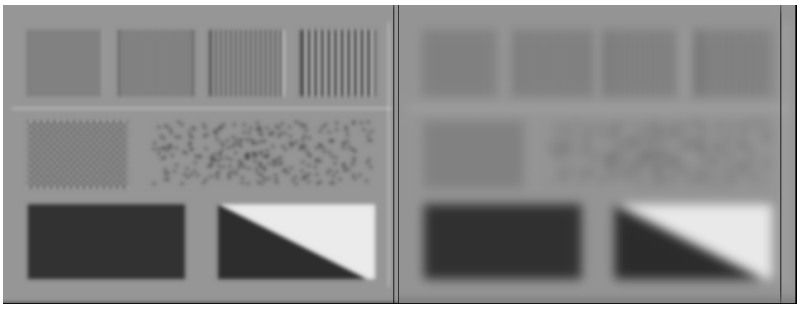


Figure 3.1: Synthetic test scene (480×360). Top row: four groups of vertical stripes with periods of 2/4/6/8 px; middle row: a 4 px checkerboard and a text-like speckle block; bottom row: a step-edge dark rectangle and a diagonal edge; one horizontal and one vertical 1 px bright line run across the frame.



(a) Moderate defocus ($\sigma = 2.4$) (b) Heavy defocus ($\sigma = 6.4$)

Figure 3.2: Two defocus levels simulated by Gaussian blur. (a) $\sigma = 2.4$: the 2/4 px fine stripe groups have melted into gray blocks, while the coarse stripes and large structures remain; (b) $\sigma = 6.4$: all detail is gone, leaving only large patches of light and dark.

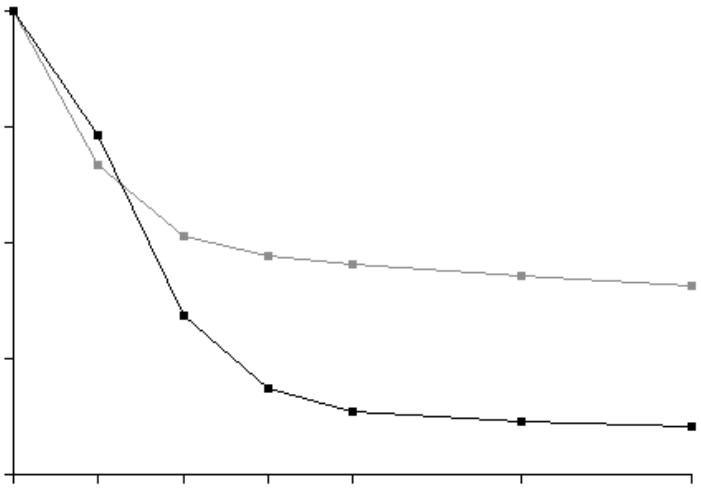


Figure 3.3: Defocus response curves of the two focus measures (horizontal axis: blur σ; vertical axis: normalized score; black: gradient method, gray: variance method). Both curves decrease strictly monotonically, but the gradient method falls to 0.105 while the variance method plateaus at about 0.41.

3.5 SciVision Implementation

Focus measures are provided in the SciVision SDK by the `SCIMV::SciSvFocus` class, whose core interface is `GetFocusScore`:

```
SCIMV::SciSvFocus focus;
SciROI roi;
SciPoint tl(0, 0), br(W - 1, H - 1);   // GenRect1 takes non-const references, so named variabl
roi.GenRect1(tl, br);

double score;
long rc = focus.GetFocusScore(frame, roi, SCI_GRADIENT, &score);
if (rc) { /* non-zero is an error code and must be checked */ }
```

An engineering detail: the Gaussian filter of the SDK used in this experiment does not fix its internal floating-point summation order, so on a rerun the 3rd significant digit of the scores jitters slightly (at $\sigma = 0$ no filtering is applied and the scores match digit for digit), but the monotonicity and discrimination conclusions are stable. With evaluation-type metrics, engineering judgment should rest on trends and orders of magnitude, not on the last digit.

The four parameters are, in order: `srcImage`, the image to be evaluated; `ROI`, which restricts the evaluation region and supports three types — axis-aligned rectangle, rotated rectangle, and undefined, where the undefined type means the whole image is processed (on the production line the ROI is usually framed around the feature under test to keep the background from contaminating the score); `method`, which selects the evaluation algorithm, see the table below; and `score`, which returns the result. The score is a relative quantity: it is only meaningful to compare scores obtained on the same scene, the same ROI, and the same method.

Table 3.2: The method parameter of `GetFocusScore`

Enum value	Method	Characteristics per the manual
`SCI_VARIANCE` (0)	Variance method	Sensitive to noise, shortest runtime
`SCI_EVA_IMPROVE`	Point-sharpness method	Fairly sensitive to noise and missing detail, longest runtime

Enum value	Method	Characteristics per the manual
`SCI_IMAGE_DIFFERENCE` (2)	Difference method	Fairly sensitive to noise and missing detail, longer runtime
`SCI_GRADIENT` (3)	Gradient method	Less affected by noise and missing detail, longer runtime

This chapter's experiments use the `SCI_GRADIENT` and `SCI_VARIANCE` settings, corresponding to the two formulas of Section 3.4. The SDK also provides `ScoreStatistics` and `CurveFitting`, for finding the highest point in a score sequence and for optimally fitting the score curve respectively — ready-made building blocks for the autofocus "coarse sweep — fine sweep — peak fitting" procedure. The complete project that generates all of this chapter's images and score tables is located at `code/cameras_and_lenses/`; readers can modify the scene structure and blur parameters to reproduce the results themselves.

Industry Case: Lens Assembly Acceptance on the Line

After a module maker's inspection equipment was duplicated onto a second production line, the false-detection rate was noticeably higher. The investigation traced it to assembly technicians focusing by eye: "looks about sharp" is simply not reproducible across different people and different monitors. The remedy was to write the focus score into the alignment/assembly process: after each unit is assembled, it photographs a uniform standard target and is scored with `GetFocusScore` (gradient method); only units reaching at least 95% of the benchmark unit's score are released. Two pitfalls are worth recording. First, the focus score is highly sensitive to the imaged content — swap in a different target sheet and the scores lose all comparability, so **acceptance must use a standard target of the same model in the same pose**. Second, the last digits of the score exhibit floating-point jitter, which is entirely absorbed

once the threshold carries a 5% margin — an acceptance criterion should never have depended on the third decimal place anyway. Once the procedure was locked in, the new line passed alignment on the first attempt, and aging units with focus drift were automatically caught by periodic re-inspection.

3.6 Summary

- **The upper bound on image quality is fixed on the optics side**, and no downstream algorithm can recover what is lost; selection calculations start from the pinhole model — $m = h/\mathrm{FOV}$, $f = m \cdot \mathrm{WD}/(1+m)$ — and then check the object-plane size each pixel covers.
- **Calculate resolution from the requirement; do not buy it by budget**: the smallest feature must cover at least 2 pixels (3–4 with margin); on a sensor of the same size, smaller pixels mean worse full well capacity and dynamic range. **Moving workpieces require a global shutter.**
- **Depth of field** $\propto cN(m+1)/m^2$: trade aperture for depth of field; at high magnification the depth of field collapses sharply. An ordinary lens's magnification varies with object distance ($\Delta m/m \approx \Delta Z/Z$), so high-precision dimensional measurement requires a telecentric lens, whose aperture must be no smaller than the measured object.
- **Sharpness can and should be quantified**: the gradient-energy method measures high-frequency detail and discriminates far better than the gray-level variance method, which sees only global contrast (a tail-end gap of 0.105 versus 0.407 in this chapter's experiment); autofocus is a coarse-to-fine hill-climbing search on the focus curve.
- **Focus scores are relative quantities**: comparable only on the same scene, the same ROI, and the same method; when used as an acceptance criterion, fix a standard target and leave threshold margin for floating-point jitter.

For systematic selection of cameras, lenses, and illumination, and a more complete optical analysis of telecentric imaging, see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018); for the geometric derivations of the pinhole model, thin-lens imaging, and perspective projection, see Szeliski's textbook (Szeliski 2022). This chapter compared only two focus measures — the gradient-energy and gray-level variance methods; to select among a much larger operator set, Pertuz et al. (Pertuz, Puig, and Garcia 2013) evaluate more than thirty focus-measure operators within a unified framework for robustness to noise, window size, and image content, a practical reference when picking a focus measure for autofocus or shape-from-focus.

4 Illumination

There is an old saying in the machine vision industry: “Half of a vision project’s success or failure is decided by the lighting.” This is no exaggeration: cameras and algorithms deal in gray values, and gray-value differences are the product of the interaction between illumination and the surface. A scratch, a missing piece of material, a row of characters — each only images as a **separable gray-value difference** under the right illumination. If the lighting fails to pull the target apart from the background, no algorithm, however ingenious, can make bricks without straw. Conversely, one well-chosen lighting setup often reduces the algorithm from elaborate texture analysis to a single threshold segmentation.

This chapter answers two questions. First, faced with a new inspection target, how do you choose the illumination — bright field or dark field, front lighting or backlight, which color? Second, what are the consequences of non-uniform illumination, and what are the two remedies: flat-field calibration on the hardware side and shading correction on the software side. The second half comes with a complete quantitative experiment.

4.1 Illumination Modes and Selection

The core variable of an illumination mode is **the relationship between the light’s angle of incidence and the lens’s viewing angle**. If the specular reflection from the surface enters the lens directly, the background is lit up — this is **bright-field illumination**. If only the light scattered by surface relief enters the lens, the flat background stays dark while raised or recessed features light up — this is **dark-field illumination**. The same scratch appears as a faint dark streak on a bright background in bright field, but as a bright thin line on a black

background in dark field — the contrast can differ by an order of magnitude. Classified by position along the optical path, there is also **front lighting**, illuminating from the camera side, and **backlight**, illuminating from behind the target; the latter images the target's outline as a clean black silhouette. Common modes and their typical applications are summarized below.

Illumination mode	Working principle	Suitable surfaces and defect types
Bright-field front light	Reflected light enters the lens; background is bright	Printing, characters, stains on diffuse flat surfaces
Low-angle ring light / dark field	Grazing-angle illumination; only scattered light from relief enters the lens	Scratches, burrs, dents, edge chipping
Backlight	Outline imaged as a black silhouette	Outer dimensions, hole diameters, lead-pitch measurement
Coaxial light	A half-mirror sends light out along the optical axis	Highly reflective flat surfaces: glass, wafers, polished metal
Diffuse dome	All-angle diffuse light, eliminating highlights	Curved and uneven metal: can bottoms, solder balls

Color is a design variable too. In monochrome inspection, a **complementary color is often used to boost contrast**: red printing appears nearly black under green light, yet all but vanishes under red light. Industrial sites favor red LEDs — not only because the devices are mature and cheap, but because paired with a narrow-band filter they effectively **reject ambient light**: shop-floor ceiling lights and daylight through skylights are all filtered out, and the image is determined solely by the controlled source. Incidentally, pushing the dark-field

idea of "using lighting direction to reveal surface relief" to its extreme yields photometric stereo, which uses sources from multiple directions to solve back for the surface normals (Chapter 34).

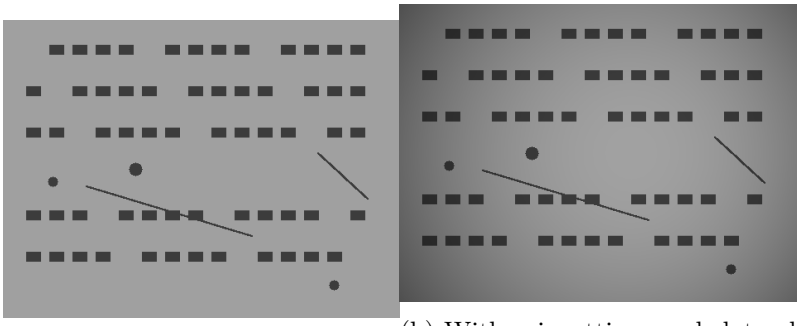
4.2 The Harm of Non-uniform Illumination

Even with the right illumination mode chosen, the **spatial uniformity** of the illumination can still go wrong. Non-uniformity has two major sources. The first is **vignetting**: a lens's natural light falloff follows the $\cos^4\theta$ law — illuminance decays with the fourth power of the **cosine** of the off-axis angle ($\cos^4\theta$) — and, combined with mechanical obstruction by the lens barrel, the corners of the frame are inherently darker than the center. The second is **source geometry**: unequal distances from the source to the surface, single-sided lighting, and unevenly aged LED beads all superimpose lateral brightness gradients. The result is that the same kind of surface shows different gray values at different positions in the image — which breaks precisely the implicit assumption behind most algorithms.

The lighting-experiment method: run a sample trial before choosing an algorithm. When a new workpiece arrives, the first thing to do is bring several typical light sources (ring, bar, coaxial, backlight) and test-shoot the sample with each one, picking the combination with the highest defect contrast and the cleanest background — only then start thinking about algorithms. An hour of experiments at the illumination stage often saves a month of struggle at the algorithm stage.

We quantify this with a synthetic experiment. The ideal scene is 480×360 pixels: on a uniform background of gray value 160, there are five rows of text-like dark bands, three defect blobs, and two thin scratches 2 px wide, all targets at gray value 60. We then overlay a synthetic non-uniform illumination field: a radial falloff toward the corners of $1 - 0.45(r/r_{\max})^2$ (45% light loss at the corners), multiplied by a horizontal gradient from 0.9 to 1.1 simulating a single-sided source bias. The two images are shown in Figure 4.1.

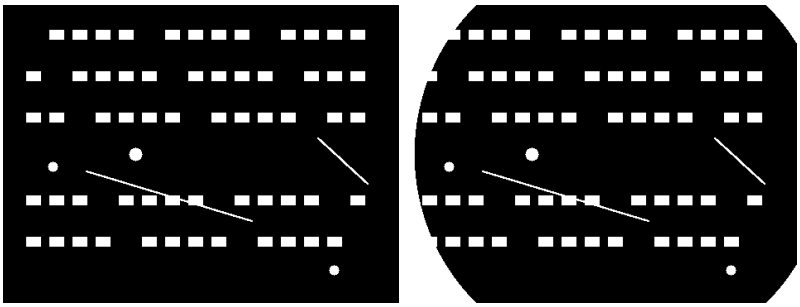
Note that to the human eye Figure 4.1b does not look too bad — the targets are still clearly visible. But an algorithm is not a human eye. With background at 160 and targets at 60, take the midpoint $T = 110$ as a fixed threshold (pixels darker than T classified as target): on the ideal image the misclassified pixels number **0**; on the vignetted image, the misclassified pixels soar to **16327** — about 9.4% of the image's 172800 pixels. The results are shown in Figure 4.2.



(a) Ideal uniform illumination

(b) With vignetting and lateral gradient

Figure 4.1: The same scene imaged under two illuminations. (a) Ideal illumination: the background is 160 everywhere, the targets 60 everywhere; (b) synthetic non-uniform illumination: 45% light loss at the corners plus a horizontal gradient — the left side and the four corners are visibly darker, yet all target structures remain discernible to the eye.



(a) Ideal image binarized: 0 misclassified

(b) Vignetted image binarized: 16327 misclassified

Figure 4.2: Binarization with the fixed threshold $T = 110$ (white = classified as dark target). (a) Under ideal illumination the segmentation is perfect; (b) under non-uniform illumination, whole patches of background in the corners drop below the threshold and flip to white, and the real targets near the corners are drowned in the misclassified regions, no longer extractable as independent connected components.

The failure mode deserves a close look: the two left corners are pressed down by the gain field to about 88 and the two right corners to about 106 — all below the threshold of 110, flipping wholesale into "target"; meanwhile the real marks lying in the corners are swallowed by these misclassified regions and can no longer be separated out. This is not a gradual degradation of "accuracy dropping a few percentage points" but **total failure** — the global threshold's premise, "the same kind of surface has the same gray value everywhere," is destroyed outright by the illumination. There are two classes of remedy: on the algorithm side, switch to locally adaptive thresholds (see Chapter 7); on the illumination side, cure the non-uniformity itself. This chapter takes the second route.

4.3 Flat-Field Calibration and Software Correction

There are two routes to curing non-uniform illumination, corresponding to the "with reference" and "without reference" working conditions.

Route one: flat-field calibration. Place a uniform white board (or a diffuse reflectance standard) on the line and capture one image — the board itself has the same gray value everywhere, so the light-and-dark variation in the captured image is purely the **gain field** of the illumination and the lens. In this experiment we apply the same gain field to a blank target of gray value 200; the resulting flat-field reference image is shown in Figure 4.3.

With the flat-field reference image F, correction is a single per-pixel division:

$$g_{\mathrm{corr}}[n, m] = \frac{g[n, m]}{F[n, m]}\,\bar{F},$$

where $\bar{F}$ is the mean of F; multiplying it back keeps the corrected image at its original overall brightness level. Once the gain field is divided out, the illumination non-uniformity vanishes at its source. Flat-field calibration is the production line's

Figure 4.3: Flat-field reference image: imaging a uniform white board, the resulting brightness distribution is the pure gain field of the illumination system — dark on the left and bright on the right, dark in the corners and bright in the center, exactly the same structure as Figure 4.1b.

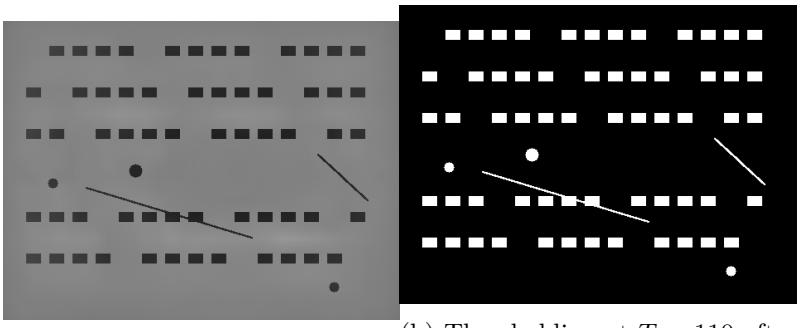
first choice: physically rigorous and computationally cheap. Its cost is that it **requires a reference image** — it must be redone after a lens change, an aperture change, or light-source aging — and some stations (continuous in-line motion, large fields of view) simply cannot accommodate a white board.

Route two: shading correction with self-estimated background. When no reference image is available, the illumination distribution can be estimated from the image itself: illumination variation is **large-scale** (a slow drift spanning the whole frame), while target structures are **small-scale** (characters a dozen-odd pixels tall, scratches a few pixels wide) — the two are separable in spatial scale. Low-pass filter at a sufficiently large scale (echoing Chapter 6, except that here the low-frequency component is treated as the "illumination layer" to be removed rather than the signal to be kept) so that the small targets are flattened away, and what remains is an estimate of the background illumination; then normalize the original image against this background, and the correction is done. The precondition follows immediately: **the targets must be significantly smaller than the scale of the**

A more rigorous flat-field calibration also subtracts the **dark frame** — capture one image with the lens cap on to obtain the sensor's fixed offset: $g_{\text{corr}} = (g - D)/(F - D) \cdot \overline{(F - D)}$. For industrial 8-bit cameras the offset is tiny and often omitted; for scientific-grade cameras and long exposures it must not be.

illumination variation — if the defect itself is a large gradual patch (such as a large-area color deviation), it will be wiped out along with the illumination.

We use the SciVision SDK's shading correction (parameters detailed in Section 4.4) to process Figure 4.1b; it is exactly an implementation of self-estimated background and **needs no flat-field reference image**. The results are shown in Figure 4.4.



(a) Image after shading correction (b) Thresholding at $T = 110$ after correction

Figure 4.4: The effect of software shading correction. (a) The background is flattened to a uniform level of about 125, with no visible brightness difference left between the corners and the center; (b) the same fixed threshold $T = 110$ works again — the segmentation result is pixel-for-pixel identical to the ideal image, with 0 misclassifications.

Quantitatively, comparing the mean gray value of 20×20 background patches at the four corners and the center:

Table 4.2: Mean gray values of background patches at the four corners and the center: correction compresses a spatial range of 72 levels down to 4.5 levels

Image	Top-left	Top-right	Bottom-left	Bottom-right	Center	Range
Ideal illumination	160.0	160.0	160.0	160.0	160.0	0

Image	Top-left	Top-right	Bottom-left	Bottom-right	Center	Range
Vignetted	87.6	105.9	87.6	105.9	159.9	72.3
Corrected	125.4	125.2	125.4	124.5	129.0	4.5

The background range converges from 72 levels to 4.5 levels. Note that the corrected level is about 125 rather than the original 160 — what software correction guarantees is "flat," not "restored to the original value"; and for threshold segmentation, flat is enough. Under the same threshold $T = 110$, the misclassified pixels drop from **16327 to 0** — pixel-for-pixel identical to the segmentation under ideal illumination.

4.4 SciVision Implementation

Shading correction is provided by the `SCIMV::SciSvShadingCorrection` class; the call used in this chapter's experiment is identical to the accompanying project code:

```
SCIMV::SciSvShadingCorrection sc;
SciImage dst;
long rc = sc.CorrectShading(src, roi,
    2,      // lightOrDark=2: keep both bright and dark targets (flatten background only)
    1.0,    // gain: gain of 1, no extra contrast amplification
    15,     // extractSize: extraction size, slightly larger than the target block height (tex
    2,      // correctionMethod=2: shading correction
    0,      // filterValue: do not exclude interfering gray values
    0,      // direction=0: both X and Y directions
    5,      // filterkernelSize: background-estimation filter kernel
    &dst);
```

The meaning of each parameter and the rationale for its value are as follows.

- `lightOrDark`: specifies whether to keep targets brighter than the background, darker than it, or both; 2 means both bright and dark targets are kept, and the correction touches only the background.

An early version of this experiment reported "221 residual misclassifications after correction, all on the outermost 1 px ring" and misattributed them to the SDK leaving the outermost rows and columns unprocessed. A later audit found the root cause on the calling side: the bottom-right corner of `SciROI::GenRect1` is an **exclusive endpoint**, and the $(W-1, H-1)$ passed at the time excluded the last row and column from the ROI. After passing (W, H) to cover the full image, the residual misclassifications vanish to 0.

- `gain`: the contrast gain after correction; 1.0 means no extra stretching. It can be raised moderately when the targets' own contrast is weak.
- `extractSize`: the structure scale removed as "foreground" during background estimation; 15 is slightly larger than the 12 px text-block height, which together with the filtering is sufficient. Set it too small and the targets get mistaken for background and wiped out.
- `correctionMethod`: **must be 2 (shading correction)**. In our tuning trials we observed (that sweep is not included in the accompanying project) that 0 (median) or 1 (mean) performs only a global level normalization — the whole image is shifted to one common mean, the corners stay dark, and the spatial non-uniformity does not budge.
- `filterValue`: excludes interfering pixels of a specified gray value; 0 means no exclusion.
- `direction`: the direction of the background gradient. In the tuning trials we observed that 1 (X only) or 2 (Y only) can also flatten the numbers, but a single-direction background estimate folds the target bands of the other direction into the background, leaving visible striping artifacts across the text bands; 0 (both X and Y) is the cleanest.
- `filterkernelSize`: the smoothing kernel for background estimation; a small kernel (5) combined with both directions is already sufficient. The larger the kernel, the smoother the background — but also the larger the background-estimation bias near large targets.

The complete project that generates all of this chapter's images and statistics is located at `code/illumination/`; you can modify the gain-field strength and each parameter to reproduce the parameter-sweep conclusions above.

Industry Case: An Illumination Faceplant in Glass Scratch Inspection

A cover-glass inspection project initially used bright-field front lighting: the glass surface reflected the light evenly into the lens, scratches caused a disturbance of only about 5 gray levels, the detection rate was below thirty percent, and for a while the team tried to brute-force it with elaborate texture-enhancement

algorithms. The setup was then switched to a low-angle dark-field ring light: the flat glass surface scatters nothing toward the lens, so the background goes nearly black while scratches scatter into bright lines; the contrast rose above 80 levels, and the algorithm collapsed to a single threshold segmentation. But dark field brought a new problem of its own — at grazing angles the illuminance is extremely sensitive to distance and angle, the background at the edges of the field of view turned visibly gray, and the fixed threshold kept raising false alarms in the edge regions. Only after layering this chapter's software shading correction on top to flatten the background did the system truly stabilize. Two lessons: illumination and algorithm are one system and must be designed together; and every doubling of contrast won by the lighting saves algorithm complexity by a factor of ten.

4.5 Summary

- **Illumination determines the gray-value difference, and the gray-value difference sets the ceiling for the algorithm.** Selection rule of thumb: diffuse surfaces — bright field; scratches and burrs — dark field (low angle); outlines and dimensions — backlight; highly reflective flats — coaxial; curved glossy surfaces — dome. Run a sample trial first, then fix the algorithm.
- **Non-uniform illumination (vignetting + lateral gradient) makes a global threshold fail outright**, not degrade gradually: in this chapter's experiment, the fixed threshold's misclassifications shot from 0 to 16327 pixels, with whole patches of corner background flipping over.
- **Flat-field calibration** divides out the gain field per pixel using a white-board reference image ($g/F \cdot \bar{F}$); physically rigorous and computationally cheap, it is the first choice whenever a reference board can be placed. It must be redone after a lens change, an aperture change, or light-source aging.
- **Software shading correction** estimates the background from the image itself with a large-scale low-pass

filter and then normalizes — no reference image needed; the precondition is that target scale is far smaller than the illumination-variation scale. In this chapter's experiment the background range was compressed from 72 levels to 4.5, and misclassifications fell from 16327 to 0 (note that the bottom-right corner of `GenRect1` is an exclusive endpoint: a full-image ROI must be passed as (W, H)).

- **The parameters are not black magic**: `correctionMethod` must be set to shading correction (0/1 only do global normalization), a single-direction `direction` leaves striping artifacts, and `extractSize` must be slightly larger than the largest target — all reproducible, measured conclusions.

For a systematic treatment of illumination engineering and radiometric calibration, see further the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018). This chapter's flat-field calibration and shading correction address spatial non-uniformity, whereas the nonlinear response between gray value and scene irradiance — the camera response function — must be calibrated separately: Mitsunaga and Nayar (Mitsunaga and Nayar 1999) introduced radiometric self-calibration, recovering the response curve from a few differently exposed images without any reference radiance source, a foundation for accurate photometry and HDR imaging. To instead exploit controlled illumination actively to extract information, Woodham's (Woodham 1980) photometric stereo solves per-pixel surface normals from several images taken under known, varying light directions from a single viewpoint — the classic paradigm of "active lighting for measurement".

5 Camera Calibration

Up to this point, every result in this book has been expressed in pixels: the edge sits at column 312.46, the circle center is off by 0.8 pixels. But acceptance criteria on a production line are never written in pixels — drawings are dimensioned in millimeters, tolerances read ±0.02 mm, and the robot arm needs positions in the workpiece coordinate frame. **Camera calibration** is the bridge from "pixels" to "millimeters": it measures the camera's own imaging parameters, after which every pixel coordinate can be converted into a physically meaningful geometric quantity. Without calibration, all the measurements of Chapter 21 are merely a game of pixels; and if the calibration is done poorly, even the cleverest algorithm will report lens distortion as workpiece deformation.

This chapter answers three questions: what the imaging process of a real camera looks like mathematically (pinhole model + intrinsics + distortion); how to recover these parameters from a few images of a calibration target (the principle and workflow of Zhang's method); and the step most easily neglected — how to verify that a calibration result can be trusted. Running through the first four sections is a "ground truth known" synthetic experiment: we build a virtual camera by hand and let it photograph a calibration target (Figure 5.1), then use SciVision to calibrate the parameters back and reconcile them against the truth, item by item — because a real camera's intrinsics are nowhere to look up, only a synthetic ground truth allows item-by-item verification. The final section then switches to a **real calibration board**, grounding the opening "pixels-to-millimeters" bridge in a nine-point coordinate calibration on an actually-photographed dot target (Section 5.5).

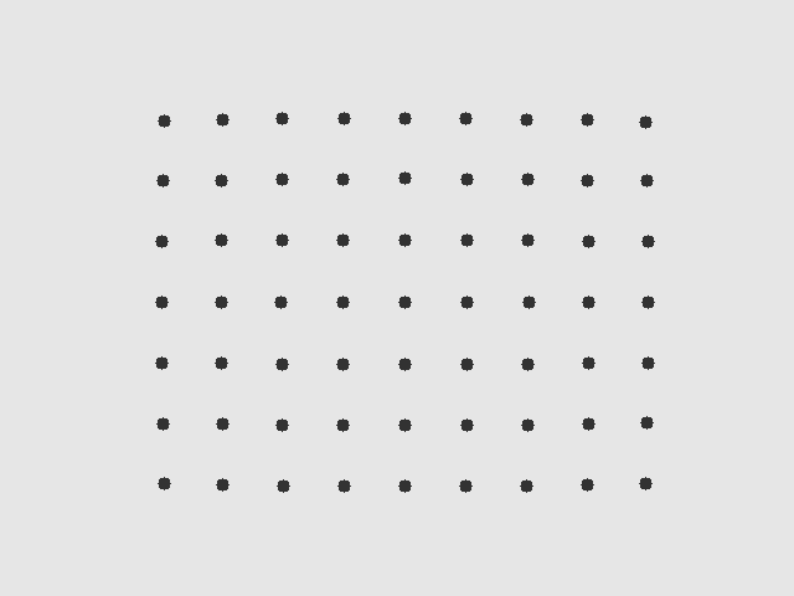


Figure 5.1: A 9×7 circle-grid calibration target photographed by the virtual camera in a fronto-parallel pose (dot spacing 10 mm; the image contains barrel distortion with $k_1 = -0.18$). The task of calibration: from a handful of such images plus the known geometry of the target's dot grid, recover all of the camera's internal parameters.

5.1 The Complete Imaging Model

Chapter 3 gave the intuition of the pinhole model: a scene point, the optical center, and the image point lie on one line. Written in matrix form with the homogeneous coordinates of Chapter 2, the projection of a world point (X, Y, Z) onto a pixel (u, v) is

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{K} \begin{bmatrix} R & \mathbf{t} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}.$$

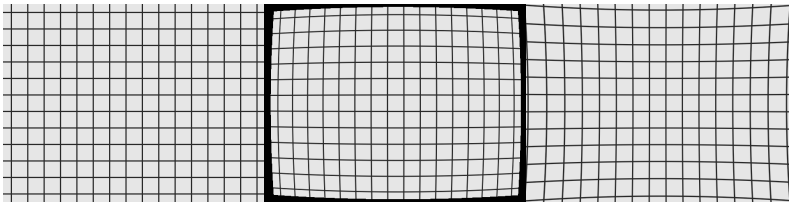
The $[R \;\; \mathbf{t}]$ on the right is the **extrinsics** — the camera's pose relative to the world coordinate frame, which changes with every new camera placement; the K on the left is the **intrinsics** matrix, determined solely by the camera and lens themselves. f_x, f_y are focal lengths in pixel units (the physical focal length divided by the pixel pitch in the horizontal and vertical directions; with square pixels the two are equal); (c_x, c_y) is the **principal point**, i.e. where the optical axis pierces the sensor, which due to assembly tolerances is usually not at the exact image center. The meaning of the intrinsics can be summed up in one sentence: f_x scales "normalized coordinates" (the tangent of the horizontal viewing angle) up into pixels, and c_x then shifts the origin to the image's top-left corner — once K is known, pixel coordinates and ray directions can be converted back and forth.

A real lens still lacks one final puzzle piece: **distortion**. The magnification of a lens assembly varies with field angle, causing image points to deviate radially from the position the pinhole model predicts. The Brown radial model describes this deviation with a polynomial in normalized coordinates:

$$x_d = x_u\,(1+k_1 r^2+k_2 r^4), \qquad y_d = y_u\,(1+k_1 r^2+k_2 r^4), \qquad r^2 = x_u^2+y_u^2,$$

where (x_u, y_u) are the ideal (undistorted) normalized coordinates and (x_d, y_d) is the actual imaged position. With $k_1 < 0$ the magnification decreases toward the field edge and straight

lines bow outward into a barrel shape; with $k_1 > 0$ they cave inward into a pincushion shape. Figure 5.2 visualizes both with a regular grid image: after applying this chapter's virtual camera distortion $k_1 = -0.18$, $k_2 = 0.05$, the "straight lines" near the edges are visibly bent. The bending can be quantified — taking 61 analytically sampled points on one horizontal grid line at the top of the image and running an orthogonal regression, the distorted points deviate from the fitted line by at most **4.678 px**, with an RMS of **2.249 px**. Compared with the 0.1 px-level subpixel localization accuracy of Chapter 14, this is a systematic error one and a half orders of magnitude larger: **without distortion correction, subpixel accuracy is meaningless**.



(a) Ideal grid (no distortion) (b) Barrel distortion ($k_1 = -0.18$) (c) Pincushion distortion ($k_1 = +0.18$)

Figure 5.2: The effect of radial distortion on a regular grid. (b) Barrel: straight lines bow outward, more strongly toward the edges; (c) pincushion: straight lines cave inward. After distortion, the top grid line deviates from a straight line by up to 4.678 px.

5.2 The Principle of Calibration

The unknowns to be estimated are f_x, f_y, c_x, c_y and k_1, k_2, plus a separate set of extrinsics for each image. The direct idea would be to photograph a known three-dimensional structure and solve backward, but 3D calibration objects are expensive to manufacture. Zhang's method (Zhang 2000) cracked the problem using nothing but a planar target, and it remains the mainstream of industrial calibration to this day. Its reasoning proceeds in three steps.

The full Brown model also includes tangential distortion terms p_1, p_2, which arise when the lens assembly and the sensor are not strictly parallel. Machine vision lenses are assembled to fairly tight tolerances, so the tangential component is typically more than an order of magnitude smaller than the radial one; this chapter's experiments therefore keep only the first two radial orders k_1, k_2 — which is also the default configuration of most industrial calibrations. SciVision's calibration output likewise reserves slots for p_1, p_2 (see Section 5.6).

Step one: each pose yields a homography. Points on the planar target satisfy $Z = 0$; substituting into the projection equation leaves only three columns of $[R\ \mathbf{t}]$, and the mapping from target plane to image degenerates into a 3×3 **homography** matrix $H = K\,[\mathbf{r}_1\ \mathbf{r}_2\ \mathbf{t}]$ (up to scale). Each image's own H can be solved from its point correspondences.

Step two: combine multiple poses to solve for initial intrinsics. $\mathbf{r}_1, \mathbf{r}_2$ are columns of a rotation matrix and must be orthonormal — this imposes two constraints on K per homography H. The intrinsics have 4 unknowns, so 3 distinct target poses already suffice for a closed-form solution; the more poses, the more stable the least-squares solution.

Step three: nonlinear refinement. The closed-form solution ignores distortion and serves only as an initial value. The final result comes from a nonlinear optimization: put the distortion coefficients and all the per-pose extrinsics into the parameter vector together, and minimize the **reprojection error** — the difference between each measured point position and the world point projected back into the image under the current parameters:

$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{M}\sum_{j=1}^{N_i}\left\|\mathbf{m}_{ij} - \hat{\mathbf{m}}(K, k_1, k_2, R_i, \mathbf{t}_i, \mathbf{M}_j)\right\|^2},$$

where $\mathbf{m}_{ij}$ is the measured pixel coordinate of target point j in image i, $\hat{\mathbf{m}}(\cdot)$ is the prediction of the complete imaging model (distortion included), and N is the total point count. This quantity is also the primary indicator of calibration quality: it directly answers "to what extent does this parameter set explain the imaging process".

5.3 A Gold-Standard Experiment: Synthetic Calibration with Known Ground Truth

Real calibration suffers from a fundamental embarrassment: **you never know the ground truth**. The calibration re-

Why must the calibration target be tilted at multiple angles? Imagine the target always facing the camera squarely: pull the camera twice as far away and double the focal length, and the resulting image is exactly the same — a fronto-parallel pose **cannot distinguish f from the distance Z**; there is no information along the focal-length direction. A tilted pose breaks this degeneracy: the amount of perspective foreshortening between the near and far ends of the target is determined jointly by f and the tilt angle, and only then does the focal length become observable. This is the plainest example of the **observability** intuition: whether a parameter can be calibrated accurately depends on whether the data contains information about it.

ports $f_x = 1226.3$ — how far is that from the true focal length? Nobody can answer: a camera's "true intrinsics" are simply nowhere to look up. The gold standard for validating a calibration algorithm is therefore a synthetic experiment: build a virtual camera with known parameters by hand, let it "photograph" the calibration target, feed the calibration algorithm as a black box, and reconcile its output against the truth item by item. This device validates not only the algorithm but also whether your understanding of the interface is correct — both engineering findings of Section 5.6 in this chapter were caught exactly this way.

The virtual camera's ground truth is $f_x = f_y = 600$ (a horizontal field of view of about 56° at 640×480 resolution), $c_x = 326$, $c_y = 243$ (the principal point deliberately offset from the image center), and $k_1 = -0.18$, $k_2 = 0.05$ (moderate barrel distortion). The calibration target is a 9×7 circle grid with 10 mm spacing, placed in 4 poses: fronto-parallel, pitched 20° about the X axis, yawed −20° about the Y axis, and a compound attitude (−15°/15°/10°). Gaussian noise with a standard deviation of 0.1 px is added to every projected point (with a fixed random seed for reproducibility), simulating the accuracy of real circle-center extraction. The 4×63=252 point pairs are fed to SciVision's `CameraCalibrate`, with the following result:

Table 5.1: Parameter recovery of the 4-pose synthetic calibration (63 points per pose, point-extraction noise 0.1 px)

Parameter	Ground truth	Calibrated	Error
f_x	600.0000	598.5707	−1.4293
f_y	600.0000	598.5468	−1.4532
c_x	326.0000	325.5661	−0.4339
c_y	243.0000	243.2207	+0.2207
k_1	−0.1800	−0.1832	−0.0032
k_2	0.0500	0.0663	+0.0163

Read the table line by line. The focal-length error is about 1.43 px, a relative error of **0.24%** — ample for the vast majority of measurement applications; the principal-point error

is under half a pixel. The reprojection RMSE is **0.1384 px**, while the theoretical floor set by the point-extraction noise is $\sqrt{2} \times 0.1 \approx 0.1414$ px: the fitting residual has settled right onto the noise level, meaning the model has **squeezed dry** every bit of systematic information in the data — what remains is purely the random noise we injected ourselves.

The only unflattering entry is k_2: an error of 0.0163, off by 33% relative to the truth. This is not a failure of the algorithm but a limitation of the data — k_2 multiplies r^4, so only points at the extreme edge of the field of view get a say in it, and in this experiment the target's projections never reached the corners of the frame. k_2 sits in a **weakly observable** state: many combinations of (k_1, k_2) are nearly equivalent over the range of r the data covers. Note that the reprojection RMSE did not suffer at all — within the region the data covers, this slightly-off parameter set and the ground truth make indistinguishable predictions. The pedagogical value of this exceeds the experiment itself: **a parameter's observability is determined by target coverage and tilt diversity** — to calibrate k_2 accurately, you must spread the target points into all four image corners. For the same reason, a calibration should be judged by its reprojection RMSE and the consistency of repeated re-calibrations, not by trusting the parameter values themselves — in a real calibration you have no truth table to check against, and the parameters are free to drift along weakly observable directions.

Why is the noise floor $\sqrt{2}\,\sigma$? Reprojection error measures the magnitude of a two-dimensional residual vector; the u and v directions each contribute a variance of σ^2, so the root mean square of the magnitude is $\sqrt{2}\,\sigma$. That the RMSE comes in slightly below this value ($0.1384 < 0.1414$) is no surprise: the model has dozens of free parameters and absorbs a small share of the random noise along the way — a normal symptom of "fitting down to the noise level". A result significantly below the noise floor, on the other hand, should raise overfitting alarms.

5.4 Distortion Correction

With the parameters in hand, the next step is to remove the distortion from the image. The correct way to rectify an image is **inverse mapping**: for every pixel (u, v) of the output (undistorted) image, use the forward distortion model to compute its source location $F(u, v)$ in the original distorted image, then fetch the gray value by the bilinear interpolation introduced in Chapter 2. The direction must not be reversed — if input pixels were "scattered" forward into the output, their landing points would not sit on the integer grid and the output image would be left with holes. Inverse mapping guarantees

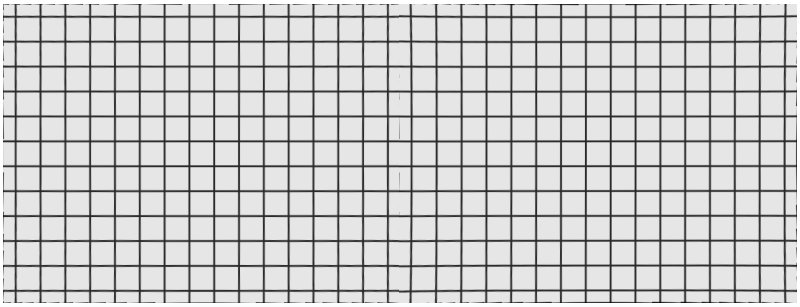
that every output pixel is assigned exactly once, at the cost of one model evaluation plus one interpolation per pixel — which can perfectly well be precomputed into a lookup table.

We first validate the implementation with the pure model: applying a hand-written inverse-mapping + bilinear-interpolation correction to the barrel-distorted grid image, the orthogonal deviation of the top grid line drops from max 4.678 px to **0.000000 px** (analytic sample points, no noise — zero to machine precision), proving that the inverse mapping and the interpolation are implemented strictly correctly. Figure 5.3a is the corrected image, with the bent grid lines pulled straight again.

SciVision takes the engineering route. Calling the single-plane distortion calibration `DistortionCalibrate` on the 63 point pairs of the fronto-parallel pose returns Rms = 0.1493; the calibration matrix it outputs then serves two purposes. The first is point correction: using `PointProject` to transform distorted pixel coordinates into corrected pixel coordinates, the collinearity deviation of the 9 points in the target's top row drops from **1.635 px** before correction to **0.121 px** — already close to the 0.1 px point-extraction noise level, with the systematic bending of the distortion essentially rooted out; fitting an affine transform from all 63 corrected points to the world grid leaves a residual RMSE of only **0.0329 mm** and a maximum of **0.0782 mm**, showing that the corrected image plane differs from the target plane by nothing more than a linear transform — from here on, pixels can be converted to millimeters by simple scaling. The second is image correction: `DistortionCorrection` directly outputs the undistorted image (Figure 5.3b), visually consistent with the hand-written implementation's result.

5.5 A Real Calibration Board: Nine-Point Coordinate Calibration

The previous four sections used a virtual camera with known ground truth for intrinsic and distortion calibration — deliberately so: a real camera's intrinsics are nowhere to look up,



(a) Hand-written inverse mapping + bilinear interpolation

(b) SDK DistortionCorrection output

Figure 5.3: Two corrections of the barrel-distorted grid. (a) Hand-written correction using the ground-truth parameters; the straightness of the grid lines is restored to machine precision. (b) SDK correction driven by the `DistortionCalibrate` result, with consistent output. The slight nibbling at the edges comes from the distorted image having no data beyond its borders.

and only a synthetic ground truth allows item-by-item reconciliation (Section 5.3). But the chapter's opening "pixels-to-millimeters" bridge has another, more direct realization that can be run end-to-end on a **real calibration board** — **nine-point coordinate calibration**. It does not estimate intrinsics; instead it uses a handful of known points on a planar target to fit the planar transform from "pixel coordinates" to "the target coordinate frame", the form of calibration most commonly used in robot hand-eye alignment (SDK manual 4.2).

The sample image of this section is taken from the "nine-point calibration" example solution bundled with Smart3: a 3×3 dark-dot target on a 1571 × 1053 grayscale image (Figure 5.4). The workflow has two steps. **Extraction**: we first try the SDK's circle-board auto-detection `FindImageCorners`(gridType=1), but on such a sparse target of only 9 points it returns 122101001 (detection failure) — auto-detection is designed for dense checkerboard/circle arrays and degrades when points are too few; so we switch to deterministic connected-component centroids of the dark blobs,

stably obtaining 9 circle centers with an average grid spacing of 236.2 px (rows 236.19, columns 236.23, indicating the target is nearly fronto-parallel with virtually no perspective). **Calibration**: feeding the 9 pixel centroids together with the target's unit-grid coordinates $(-1, 1) \dots (1, -1)$ (each cell is 1 world unit) to `PointCalibrate` yields the 3×3 planar calibration matrix

$$M = \begin{bmatrix} 0.004233 & \approx 0 & -3.0255 \\ \approx 0 & -0.004234 & 2.3963 \\ 0 & 0 & 1 \end{bmatrix},$$

with pixel scales $s_x = 0.0042332$, $s_y = 0.0042338$ world units per pixel (i.e. 236.2 px corresponds to 1 grid cell). Projecting the 9 points back to the target frame through M, the fitting residual RMS is only **0.000106 world units** (about **0.025 px**), with a maximum point deviation of 0.000147 world units (about 0.035 px). This target is a software-rendered ideal dot array with no lens distortion, so a residual hugging the subpixel extraction accuracy is to be expected — its value lies in demonstrating the complete "pixels-to-coordinate-frame" closed loop on a real image.

More convincing is the cross-validation: the example solution ships with the calibration result the SDK's graphical interface originally computed (`calib_data.xml`), whose matrix is $a_{11} = 0.004233$, $a_{22} = -0.004236$, $t_x = -3.027629$, $t_y = 2.398271$ with scales $(0.0042328, 0.00423579)$. Extracting points and calibrating **independently in code**, our linear coefficients and scales agree with these to the 10^{-6} level, with the translation terms differing by about 0.002 world units due to the different extraction method (the interface rounds circle centers to the half-pixel) — our hand-written pipeline reproduced the result of the commercial interface, another way of grounding interface understanding in practice.

Why keep intrinsic/distortion calibration synthetic yet use a real board for coordinate calibration? The two are accepted in fundamentally different ways: the gold standard for intrinsic calibration is reconciliation against **ground truth**, which is unknowable for a real camera and hence must be synthetic (Section 5.3); nine-point coordinate calibration produces merely a "pixels-to-coordinate-frame" planar transform, whose quality is accepted via the **fitting residual** and **reproducing a known solution**, independent of any camera ground truth — so it can, and should, be

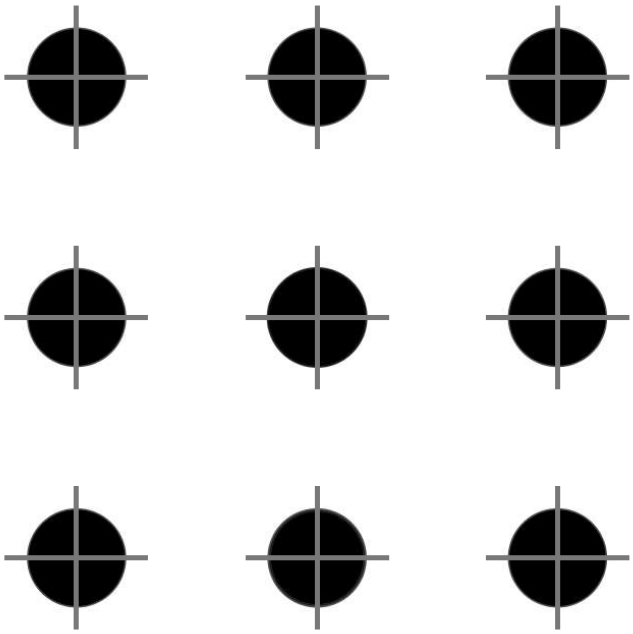


Figure 5.4: The real 3×3 dot target from Smart3's "nine-point calibration" example solution (`CalibImage/0.bmp`). Gray crosses mark the 9 circle centers from hand-written centroid extraction; these 9 points fit the planar transform from pixels to the target coordinate frame via `PointCalibrate`, reproducing the SDK interface's calibration result bundled with the example.

5.6 SciVision Implementation

Multi-pose intrinsic calibration is performed by `SCIMV::SciSvCalibration::CameraCalibrate`, corresponding to the Zhang workflow of Section 5.2. The point correspondences of all poses are concatenated in order into two arrays, and an index array declares the point count of each view:

```
SCIMV::SciSvCalibration calib;
SciPointArray allImg, allObj;   // image/world points of the 4 poses, concatenated in order
SciVarArray ptsIndex;           // point count per pose (four 63s in this example)
for (int p = 0; p < 4; ++p) {
    for (size_t k = 0; k < wX.size(); ++k) {
        SciPoint ip(imU[p][k], imV[p][k]), op(wX[k], wY[k]);
        allImg.Append(ip); allObj.Append(op);
    }
    SciVar n((long)wX.size());
    ptsIndex.Append(n);
}
SciMatrix camMat;
double rmsC = -1;
long rc = calib.CameraCalibrate(allObj, allImg, ptsIndex,
                                /*W*/ 640, /*H*/ 480, &camMat, &rmsC);
```

`allObj` holds the target-plane coordinates (in mm; the planar target has $Z = 0$, so only 2D points are filled in), `allImg` the corresponding pixel coordinates; `ptsIndex` slices the flat arrays back into individual views, the image width and height are used for the intrinsics' initial values, and the outputs are the parameter matrix `camMat` and the reprojection RMSE. Every number in Table 5.1 comes from this single call.

In use, this interface exposed two documentation problems, recorded here verbatim — they are exactly the situations engineers genuinely run into when integrating a commercial SDK. **First, the element layout of the `cameraParam` matrix is not documented in the header.** We reverse-engineered the actual layout from the synthetic experiment's ground truth: row 0 is $[\_,\ f_x,\ c_x,\ f_y,\ c_y]$ and row 1 is $[k_1,\ k_2,\ p_1,\ p_2,\ \_]$ — without a truth table to check against, reading f_y and c_x

swapped would not necessarily be noticed right away. **Second, the behavior of `PointProject` does not match the header's description.** The documentation says it returns world coordinates, but when used with a distortion calibration matrix, what it actually returns is **corrected pixel coordinates**. Following the documentation at first, we subtracted its output directly from the world grid (in mm) and got an absurd "residual" of 427 mm; only after tracing the output's units did we redesign the validation into the "fit an affine transform from corrected points to the world grid" scheme of Section 5.4. Both findings point to the same engineering habit: **your understanding of an interface must be validated on data with known answers before it is used on a production line where the answers are unknown** — which is the other dividend of the gold-standard experiment.

The real-board coordinate calibration of Section 5.5 uses another interface of the same module, `PointCalibrate(imagePoint, worldPoint, type, …)`: it takes $N \geq 4$ pairs of pixel and target points and outputs a 3×3 planar transform matrix, the pixel scales s_x, s_y, and the fitting residual, with `type=0/1/2` selecting planar / tilted / similarity (Helmert) calibration; when auto-detection fails, deterministic centroid extraction supplies the input side. The `SciSvCalibration` module also covers extended scenarios such as alignment/registration and hand–eye calibration (SDK manual 4.3): only by relating the camera coordinate frame to the motion platform's coordinate frame can calibration results drive a robot arm. These belong, together with the 3D measurement of Chapter 30, to the downstream of the "calibration chain" and are not expanded here. The complete runnable project that generates all of this chapter's experimental images and numbers lives in `code/camera_calibration/`.

Industry Case: Why Calibrations "Expire"

A structural-part measurement station passed acceptance smoothly at commissioning; three months later, a sampling check uncovered a systematic deviation of 0.03 mm. The diagnosis: heat buildup on the line expanded the lens barrel and shifted the focal plane, changing the effective focal length; vibration from the conveyor, meanwhile, rotated the

camera bracket ever so slightly — intrinsics and extrinsics had both "expired", with not a single line of the program changed. The countermeasure has two layers. One is **periodic recalibration** — redo the full calibration quarterly or at major maintenance milestones. The other, far cheaper, is a **routine check**: fix a reference standard of known dimensions in the field of view, measure it once per shift, and trigger recalibration only when a threshold is exceeded. Verification is far cheaper than calibration: one measurement, checked against one number, answers "can the calibration still be trusted". In high-accuracy settings this check belongs on the equipment's routine-check sheet — calibration parameters are not a software configuration but a physical measurement result that slowly drifts with temperature and vibration.

5.7 Summary

- **The complete imaging model = pinhole + intrinsics + distortion**: the intrinsics matrix converts ray directions into pixels, and the Brown radial model $x_d = x_u(1 + k_1 r^2 + k_2 r^4)$ describes a real lens's deviation from the pinhole. This chapter's barrel distortion of $k_1 = -0.18$ bent straight lines by 4.678 px — without distortion correction, subpixel localization is meaningless.
- **Zhang's method runs "homography → closed-form initial value → nonlinear refinement"**: each pose of the planar target contributes one homography and two intrinsic constraints; multiple poses jointly solve the initial value, and the final step minimizes the reprojection error. The target must be tilted at multiple angles — a fronto-parallel pose cannot distinguish focal length from distance.
- **The synthetic gold-standard experiment is the gold standard for validating a calibration**: only with known ground truth can you reconcile item by item. In this chapter's experiment, the focal-length error was 0.24% and the reprojection RMSE of 0.1384 px hugged the $\sqrt{2}\sigma = 0.1414$ px noise floor — the information was squeezed dry; the outsized k_2 error, meanwhile, exposed

the weak observability caused by insufficient target coverage.

- **Judge a calibration by reprojection RMSE and repeat consistency, not by the parameter values themselves**: real calibrations have no ground truth to check against, and parameters will drift along weakly observable directions; image rectification uses inverse mapping + bilinear interpolation, and correction quality can be accepted quantitatively via geometric invariants such as collinearity ($1.635 \rightarrow 0.121$ px).
- **Interface understanding must pass the gold-standard experiment too**: the parameter-matrix layout and the output units of `PointProject` were both confirmed only by checking against ground truth — when documentation and implementation disagree, data with known answers is the only arbiter.
- **What runs end-to-end on a real board is nine-point coordinate calibration**: with hand-written centroid extraction on Smart3's actually-photographed 3×3 dot target (auto-detection fails on the sparse 9-point target), `PointCalibrate` fits the pixels-to-target planar transform with a residual of about 0.025 px, and its linear coefficients and scales reproduce the bundled SDK-interface solution to the 10^{-6} level — intrinsic calibration is verified against synthetic ground truth, coordinate calibration against real data and reproduction of a known solution, each as appropriate.

For a systematic treatment of camera calibration (including richer distortion models, uncertainty analysis, and 3D calibration), read further in the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018); Zhang's original paper (Zhang 2000) is itself a very readable text. The modern treatment of distortion and intrinsics traces back to two classics: Brown (D. C. Brown 1971) gave, in close-range photogrammetry, the radial/tangential distortion model still in use today — this chapter's k_1, k_2, p_1, p_2 originate from it — while Tsai (R. Y. Tsai 1987) proposed a two-stage technique for high-accuracy 3D metrology from a single target image, the industry mainstream before Zhang's method. The extended scenario of relating the camera frame to a motion platform (SDK manual 4.3) is

precisely hand-eye calibration, whose foundational algorithm is given by Tsai and Lenz (R. Y. Tsai and Lenz 1989), solving the rigid transform between camera and end-effector from imaging at several known robot poses.

Part III

Image Enhancement

This part covers algorithms that improve image quality and prepare for subsequent locating and measurement: spatial filtering, thresholding, morphology, gray-level transforms, geometric transforms, frequency-domain processing, and advanced enhancement.

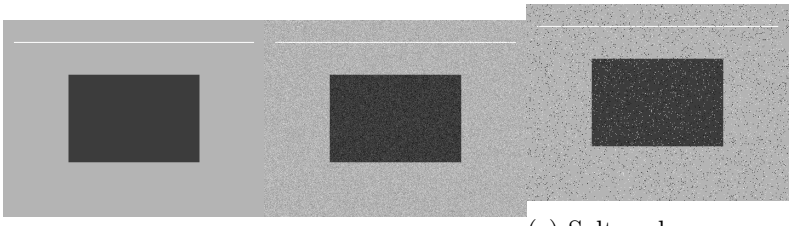
6 Spatial Filtering

Point an industrial camera at a stationary, uniformly lit gray board, capture two consecutive frames, and subtract them. The difference is not zero: every pixel jitters randomly within a small range. This is image noise. It comes from the sensor's dark current and readout circuitry, from the analog gain raised to compensate for insufficient illumination, from lighting fluctuations caused by power-supply ripple, and sometimes from dust on the lens or protective glass. In the lab these disturbances may be harmless, but on a production line, noise directly undermines everything downstream: edge localization fires false responses at noise, thresholding sprouts patches of fragments, and the repeatability of subpixel measurement degrades accordingly. This is why **spatial filtering is almost always the first step before localization and measurement** — a well-chosen "averaging" over each pixel's neighborhood that suppresses random disturbances while preserving, as much as possible, the structures we actually care about.

Throughout this chapter we use one synthetic test scene for all experiments: a dark rectangle on a uniform bright background (providing step edges), with a thin bright line only 2 pixels wide above it (simulating a fine structure such as a scratch or a bonding wire). Figure 6.1 shows the original scene and what it looks like under two typical kinds of noise.

6.1 Image Noise Models

To choose the right filter, you first have to understand what the noise looks like. The most common model in machine vision is **additive Gaussian noise**: the observed image g equals the true image f plus a random disturbance,



(a) Original scene (b) Gaussian noise (c) Salt-and-pepper noise

Figure 6.1: The synthetic test scene and two typical kinds of noise. (a) Clean scene: a bright background, a dark rectangle, and a thin bright line 2 px wide; (b) with additive Gaussian noise of standard deviation $\sigma = 18$, the whole image takes on a fine-grained texture; (c) with 4% salt-and-pepper noise, isolated pure-black and pure-white bad pixels are scattered about while the remaining pixels are untouched.

$$g[n, m] = f[n, m] + \eta[n, m], \qquad \eta \sim \mathcal{N}(0, \sigma^2).$$

Every pixel is corrupted, but usually not by much — most of the disturbance falls within $\pm 2\sigma$. Its physical origins are the sensor's dark-current noise, readout noise, and the electronic noise introduced by amplifier gain. These factors are the superposition of many small independent disturbances, and by the central limit theorem their sum tends toward a Gaussian distribution. Figure 6.1b shows exactly this situation — the image is "fuzzy" everywhere, but no single pixel is destroyed outright.

The other common kind is **salt-and-pepper noise**, also called impulse noise: a small fraction of pixels are randomly replaced by the minimum value ("pepper", 0) or the maximum value ("salt", 255), while the remaining pixels are entirely unaffected. Its sources include dead sensor pixels, transmission errors between the camera and the industrial PC, and isolated overexposed spots caused by specular reflection on highly reflective surfaces. The scattered black and white dots in Figure 6.1c are typical salt-and-pepper noise — note that a corrupted pixel bears no relation whatsoever to its true value; the error can span the entire gray-level range.

The difference between these two models is far more than a mathematical curiosity: it directly determines the choice of filter. With Gaussian noise, every pixel's observation is "roughly correct"; averaging the values in a neighborhood lets the independent zero-mean errors cancel, and the estimate improves — linear filtering is the natural choice. With salt-and-pepper noise, a corrupted pixel carries not "information with some error" but pure garbage; folding garbage into an average only contaminates the neighbors that were clean to begin with. What we need is a **nonlinear** mechanism that can recognize and discard outliers. The experiments later in this chapter will confirm this point again and again.

Why do the two kinds of noise call for different filters? Gaussian noise means "everyone is slightly wrong" — **averaging** the neighborhood lets the errors cancel each other out. Salt-and-pepper noise means "a few are wildly wrong" — averaging only spreads the wild values to the neighbors. The right move is to **discard** the outliers, which is precisely what the median filter does.

6.2 Linear Filtering and Convolution

The unifying mathematical form of linear spatial filtering is **convolution**. Given an input image f and a small template of weights — called the **kernel** h — the output image is

$$g[n, m] = \sum_{k,l} f[n - k,\, m - l]\, h[k, l].$$

Intuitively, the output value at (n, m) is a weighted sum of a small input neighborhood centered at that point, with the weights given by the kernel. The kernel determines the filter's entire character.

The simplest kernel is the **mean filter** kernel: a $K \times K$ window in which every weight equals $1/K^2$. For example, the 3×3 mean kernel is

$$h = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Note the normalization factor in front of the coefficients: the sum of all kernel weights is called the filter's **DC gain**. Normalizing the weights to sum to 1 means that filtering an image of constant gray value returns it unchanged — the average

brightness of the image is preserved. This matters especially in measurement applications: if filtering quietly raised or lowered the overall gray level, every downstream algorithm that relies on gray-value thresholds or gray-value interpolation would be dragged along with it.

The mean kernel treats every pixel in the window equally, whereas the **Gaussian filter** assigns weights that decay with distance — the farther a pixel is from the center, the less it influences the output:

$$h[n, m] = \frac{1}{2\pi\sigma^2} \exp\Big(- \frac{n^2 + m^2}{2\sigma^2} \Big).$$

The parameter σ controls the spatial scale of smoothing: the larger σ, the wider the effective neighborhood that participates in the average, the stronger the denoising — and the blurrier the image. Compared with the mean kernel, the Gaussian kernel has a monotonically decreasing frequency response with no sidelobe ringing, so its smoothing is "cleaner"; this makes it the default choice for linear denoising. In a discrete implementation, we sample the continuous Gaussian and then divide by the sum of the samples so that the DC gain is exactly 1.

In practice there are two further engineering points. First, **the kernel size must match** σ. The Gaussian function extends infinitely in theory, but beyond 3σ its amplitude is only about 1% of the central value, so it is usually truncated at $K \approx 6\sigma$ (rounded to the nearest odd number): $\sigma = 1.2$ paired with a kernel of roughly 7×7 is sufficient. Too small a kernel clips the Gaussian's shoulders; too large a kernel simply wastes computation. Second, **the Gaussian kernel is separable**: the two-dimensional Gaussian is exactly the product of a one-dimensional horizontal Gaussian and a one-dimensional vertical Gaussian, so one 2D convolution can be split into a horizontal 1D pass followed by a vertical 1D pass with identical results.

The payoff of separability: a $K \times K$ 2D kernel costs $O(K^2)$ multiply-adds per pixel, but split into two 1D kernels it costs only $O(2K)$. At $K = 7$ that is 49 versus 14; at $K = 15$ it is 225 versus 30 — the larger the kernel, the bigger the savings. Virtually every commercial vision library implements Gaussian filtering this way internally.

6.3 Nonlinear Filtering

Linear filtering has one limitation it cannot escape: it cannot tell "noise" from "edges" — both are high-frequency content, so suppressing the former inevitably blunts the latter. Nonlinear filters step outside the weighted-average framework and thereby gain abilities that no linear method can have.

The **median filter** is the most important member of this family. Instead of computing a weighted sum, it sorts the pixel values in the window and outputs the **median**:

```
For each pixel (n, m) of the output image:
    take all pixel values in the K×K neighborhood centered at (n, m)
    sort these values in ascending order
    output ← the value in the exact middle of the sorted list (the median)
```

The median is naturally immune to outliers: if a few bad pixels of 0 or 255 land in the window, as long as they make up less than half of it, sorting pushes them to the ends of the list and they never get a chance to be the output. This is exactly the nemesis of salt-and-pepper noise. Moreover, when the median filter steps across a step edge, the majority of pixels in the window come from one side of the edge, and the output simply takes a representative value from that side — the edge stays sharp, with none of the transition band that linear filtering would smear out. The price it pays we will reveal in the experiments section.

The **bilateral filter** (Tomasi and Manduchi 1998) brings the edge-preserving idea into the weighted-average framework. The weight it assigns to each neighbor $\mathbf{q}$ is the product of two Gaussian factors:

$$g[\mathbf{p}] = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q} \in S} G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|)\, G_{\sigma_r}(|f[\mathbf{p}] - f[\mathbf{q}]|)\, f[\mathbf{q}],$$

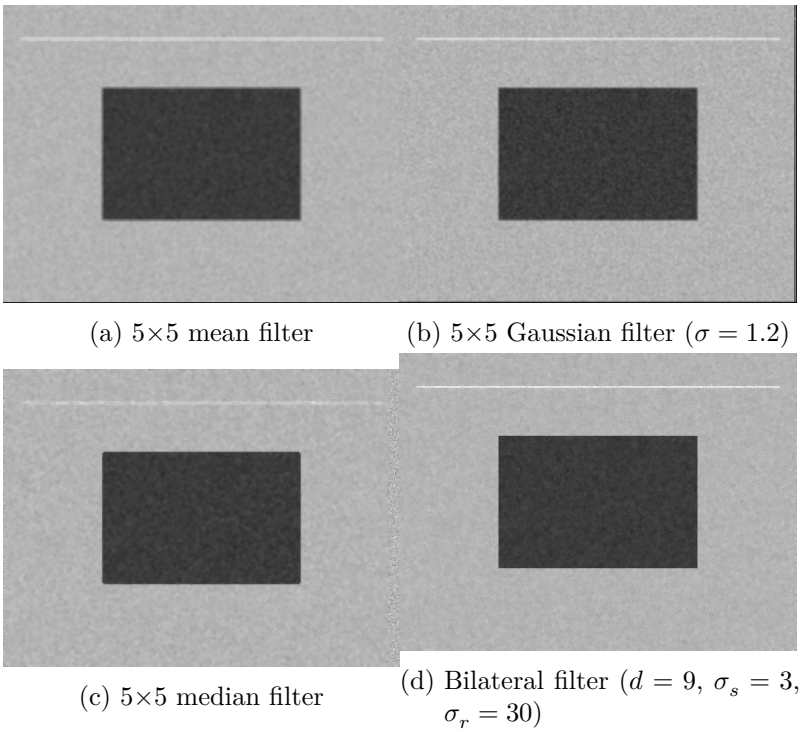
where $W_{\mathbf{p}}$ is the sum of the weights (ensuring a DC gain of 1), G_{σ_s} is a Gaussian over the spatial domain, and G_{σ_r} is a Gaussian over the gray-value (range) domain. The intuition is crystal clear: for a neighbor to have a say in the output,

it must satisfy two conditions at once — it must be **nearby** (high weight from the spatial Gaussian) **and look alike** (gray value close to the center pixel, high weight from the range Gaussian). In flat regions, neighbors have similar gray values and the bilateral filter degenerates into ordinary Gaussian smoothing, suppressing noise effectively. Near an edge, pixels on the opposite side differ greatly in gray value, the range Gaussian drives their weights toward zero, and averaging takes place only on the same side of the edge — so the edge is left untouched. σ_r draws the boundary of "one of us": neighbors whose gray-value difference is below roughly $2\sigma_r \sim 3\sigma_r$ join the average, while those beyond it are treated as "the other side" and excluded.

6.4 Experimental Comparison

Now let the four filters compete head to head. First, their performance under Gaussian noise (Figure 6.1b); the four filtered results are shown in Figure 6.2.

The mean and Gaussian filters both visibly flatten the grain, confirming the principle that "averaging cancels independent errors." The price is equally visible: the rectangle's edges are smeared into a gray transition band, and the 2 px thin bright line becomes dimmer and thicker, because its brightness has been spread across the surrounding background pixels. Comparing the two, at the same kernel size the Gaussian filter degrades the edges slightly less — a consequence of its weights being concentrated toward the center. The median filter keeps the rectangle's edges remarkably sharp, but look at the thin bright line: it has all but vanished — a phenomenon we will discuss separately in a moment. The most balanced performer is the bilateral filter: the background noise is smoothed away, the rectangle's edges stay crisp, and the thin line survives intact, because the gray-value difference between the line and the background (about 70 gray levels) far exceeds the $\sigma_r = 30$ "one of us" threshold — line pixels average only with line pixels. **For Gaussian noise, the bilateral filter is the best choice for edge-preserving denoising.**



(a) 5×5 mean filter

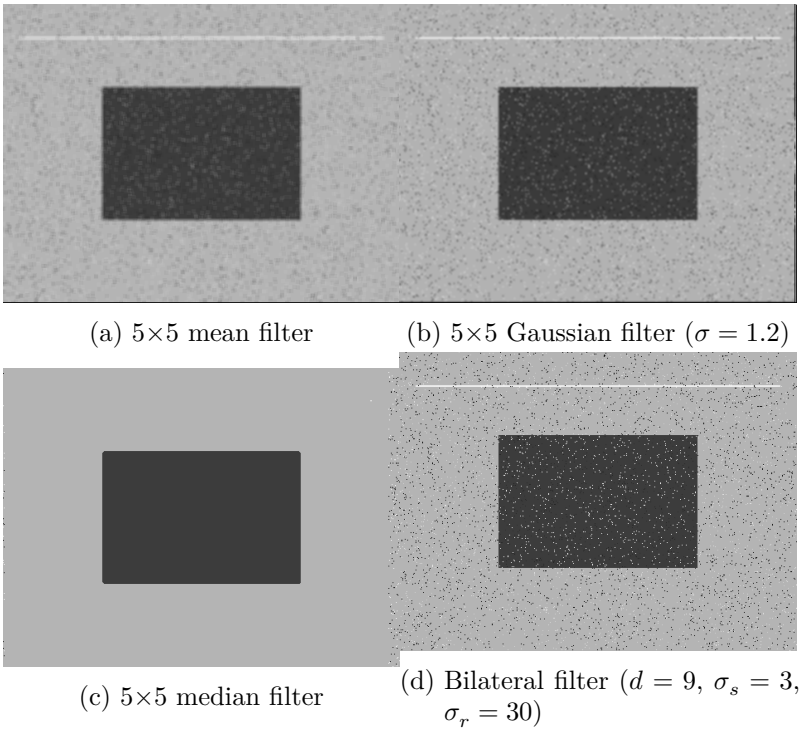
(b) 5×5 Gaussian filter ($\sigma = 1.2$)

(c) 5×5 median filter

(d) Bilateral filter ($d = 9$, $\sigma_s = 3$, $\sigma_r = 30$)

Figure 6.2: The Gaussian-noise image processed by the four filters. The mean and Gaussian filters suppress noise effectively but blunt both the edges and the thin line; the median filter preserves the rectangle's edges but nearly erases the thin bright line; the bilateral filter suppresses noise while preserving both the step edges and the thin line best.

Switch to salt-and-pepper noise (Figure 6.1c), and the conclusions all but reverse; see Figure 6.3.



(a) 5×5 mean filter

(b) 5×5 Gaussian filter ($\sigma = 1.2$)

(c) 5×5 median filter

(d) Bilateral filter ($d = 9$, $\sigma_s = 3$, $\sigma_r = 30$)

Figure 6.3: The salt-and-pepper image processed by the four filters. The mean and Gaussian filters smear each impulse into a gray blotch; the median filter removes the salt-and-pepper noise almost completely while keeping the edges sharp, but it erases the thin bright line as well; the bilateral filter is nearly useless against salt-and-pepper noise — the isolated bad pixels are protected as if they were "edges."

The mean and Gaussian results are disappointing: the impulses do not disappear — they get smeared. Each bad pixel of 0 or 255 shares its extreme value across the entire window, turning into a soft light or dark blotch about 5 px across, and the whole image ends up looking grimy. The median filter, by contrast, is overwhelmingly superior: the black and white bad pixels are removed wholesale, the background returns to uniformity, the rectangle's edges remain sharp, and the result is nearly as clean

as the original. This is the visual confirmation of the claim at the end of Section 6.1 — for outliers, discard, don't average.

The result most worth remembering is the bilateral filter's: **it is nearly useless against salt-and-pepper noise.** Compare Figure 6.3d with Figure 6.1c: the bad pixels remain almost exactly where they were. The reason is no mystery, though it is often misunderstood: a salt pixel of value 255 differs from its surrounding background of about gray level 180 by 75 gray levels, far beyond the $\sigma_r = 30$ similarity threshold. In the bilateral filter's eyes, this bad pixel is a "distinctive small structure" — none of its neighbors count as "one of us," so it averages only with itself and naturally keeps its value. **The bilateral filter's edge-preserving mechanism becomes precisely a shield for impulse noise.** When you face salt-and-pepper noise, reach straight for the median filter.

The median filter has its own bill to pay, however. Look again at Figure 6.3c and Figure 6.2c against the original in Figure 6.1a: the 2 px thin bright line has vanished in both median results. The reason is exactly the same one that removes the bad pixels — within a 5×5 window, the thin line contributes at most 2×5 = 10 pixels, fewer than half of the 25, so after sorting it can never occupy the median position, and it gets removed as "somewhat larger noise." **The median filter preserves edges, but not fine structures**: any line-like or dot-like structure narrower than half the window width is wiped out, whether it is noise or a genuine target. This yields a hard constraint: **the median kernel must be smaller than (twice) the width of the smallest structure to be preserved** — if the thin line is 2 px wide, a 3×3 median kernel can still spare its life, but 5×5 will carry it away together with the noise. The filter does not know defects from noise; it only knows size.

6.5 SciVision Implementation

The four filters of this chapter are provided in the SciVision SDK by the `SCIMV::SciSvFilter` class, invoked as follows:

```
SCIMV::SciSvFilter f;
SciImage dst;
f.Mean(src, roi, &dst, 5, 5);
f.Gaussian(src, roi, &dst, 5, 5, 1.2, 1.2);
f.Median(src, roi, &dst, 5, 5);
// Bilateral(kernel diameter 9, spatial  =3.0, range  =30.0): neighbors whose gray-value differ
f.Bilateral(src, roi, &dst, 9, 3.0, 30.0);
```

Among the common parameters of these interfaces, `src` is the input image, `roi` restricts the processing region (production-line programs usually filter only the inspection region to save time), and `dst` receives the output. The filter-specific parameters are as follows.

- `kernelSizeX` and `kernelSizeY` of `Mean` / `Median` (both 5 in the example above) are the window's width and height; they must be odd so that the window has a center. The two may differ — for example, a flat `(9, 3)` window provides directional suppression of horizontal stripe noise.
- `Gaussian` additionally accepts `sigmaX` and `sigmaY` (both 1.2 above), the horizontal and vertical Gaussian standard deviations. Remember the matching rule of Section 6.2: the window side should be roughly 6σ; pairing $\sigma = 1.2$ with 5×5 as above is a slightly tight truncation, acceptable in practice.
- The three parameters of `Bilateral` are, in order, `kernelSize` (the neighborhood diameter, 9 above), `space_sigma` (the spatial-domain σ_s, controlling the scale of "nearby"; 3.0 above together with diameter 9 satisfies the common ratio $d \approx 3\sigma_s$), and `color_sigma` (the range-domain σ_r, controlling the "looks alike" threshold; 30.0 above means neighbors differing by more than about 90 gray levels get near-zero weight, so step edges above 70 levels are preserved almost intact).

The complete runnable project that generates all of this chapter's experimental images is located at `code/spatial_filtering/`; readers can modify the noise levels and filter parameters to reproduce the results themselves.

Industry Case: Preprocessing for Scratch Detection on Metal Parts

In an inspection project for stamped metal parts, the sand-blasted workpiece surface produced a large number of isolated bright specular spots whose statistics closely resembled salt-and-pepper noise. Initially a Gaussian filter was used for preprocessing: the specular spots were smeared into soft light-gray blotches that were hard to distinguish from scratches, and the false-detection rate was high. Switching to a 3×3 median filter removed the specular spots cleanly while keeping scratch edges sharp, and false detections dropped dramatically. A later "optimization" enlarged the median kernel to 7×7 — and the line immediately began missing fine scratches: the scratches were only 2–3 px wide, less than half the window's population, and were removed as noise. This was the thin-bright-line experiment of this chapter replayed on the factory floor. Rule of thumb: **choose the kernel at 2–3 times the noise-grain diameter, and keep it smaller than twice the width of the smallest defect**; when the two conflict, preserving the defect comes first, and the residual noise is left to downstream area-based screening.

6.6 Summary

The key points of this chapter can be distilled as follows.

- **Identify the noise first, then choose the filter.** Gaussian noise is "everyone slightly wrong," suited to averaging-type linear filters; salt-and-pepper noise is "a few wildly wrong," requiring discard-type methods such as the median filter.
- **The kernel of a linear filter should be normalized to a DC gain of 1**, preserving the image's average brightness. Truncate the Gaussian kernel at $K \approx 6\sigma$, and exploit separability to cut the computation from $O(K^2)$ to $O(2K)$.
- **Bilateral filtering = spatial proximity × gray-value similarity, weighted together.** It is the best edge-preserving denoiser under Gaussian noise; but

isolated impulses are protected as if they were "edges," so it is nearly useless against salt-and-pepper noise.

- **The median filter preserves edges but not fine structures**: thin lines and small dots occupying less than half the window are wiped out together with the noise, so the kernel size must stay below twice the width of the smallest structure to be preserved.
- **Filter parameters are part of the measurement accuracy**: kernel size and σ should be chosen from the noise scale and the smallest target size — not on the principle that "bigger is cleaner."

For the systematic theory of spatial filtering (linear filters, order-statistic filters, and the link to the frequency domain), see the classic textbook by Gonzalez and Woods (Gonzalez and Woods 2018). The original construction of the bilateral filter is the paper by Tomasi and Manduchi (Tomasi and Manduchi 1998), cited several times in the body of this chapter, and the canonical edge-preserving advanced method of anisotropic diffusion originates with Perona and Malik (Perona and Malik 1990). For a more systematic treatment of smoothing filters in industrial inspection pipelines, see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

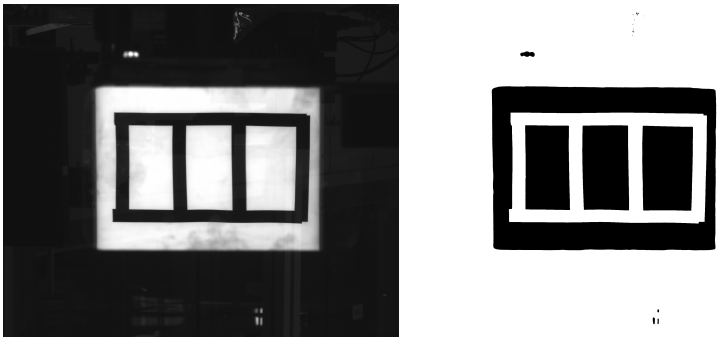
7 Thresholding

So far our object of processing has been "the image" — a two-dimensional array of gray values. But what a production line actually cares about is "the object": where are the pins of this chip, what is the diameter of this hole, is there contamination on this panel? Crossing from the world of pixels into the world of objects almost always begins with **segmentation** — separating the pixels that belong to the object from the pixels that belong to the background. And the most widely used, fastest, and most easily underestimated segmentation tool in industrial vision is **thresholding**: pick a gray value T, assign pixels darker (or brighter) than it to the foreground and the rest to the background, and output a black-and-white binary image.

Figure 7.1 shows a typical scene: a panel under backlight illumination, where a dark tape grid stands out as a crisp silhouette against the bright light guide plate. A single automatic threshold extracts the tape grid cleanly as white foreground — no sophisticated algorithm required. This is precisely the promise of Chapter 4 — "proper lighting can reduce the problem to a single thresholding step" — being cashed in. But the other side of the same coin is this: thresholding's dependence on gray values is so direct that its sensitivity to illumination changes is equally drastic. This chapter makes both sides of that coin concrete with a set of quantifiable experiments.

7.1 Global Thresholding and the Histogram

Global thresholding is thresholding in its plainest form: the entire image shares a single threshold T. Taking the extraction of dark objects as an example,



(a) Original backlit panel (b) Otsu thresholding result

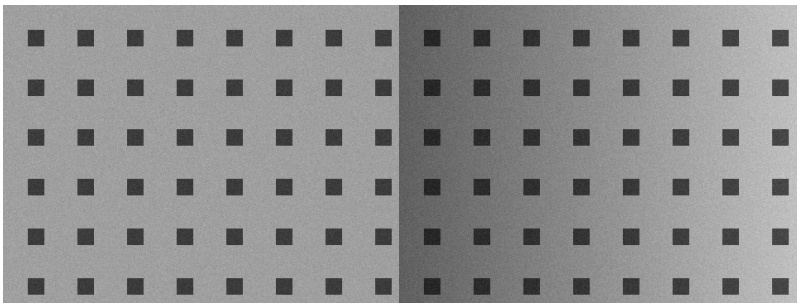
Figure 7.1: Thresholding a real image. (a) A panel under backlight illumination, with a dark tape grid set against the bright light guide plate; (b) the binarization result of Otsu's automatic threshold (which selects $T = 125$ on its own): the tape and the surrounding dark regions, whose gray values fall below T, are marked as foreground (white), and the grid structure is extracted cleanly.

$$g[n,m] = \begin{cases} 255, & f[n,m] \le T, \\ 0, & f[n,m] > T. \end{cases}$$

The more general form is **band thresholding**: given an interval $[T_{\text{low}}, T_{\text{high}}]$, pixels whose gray values fall inside the band are foreground. A single threshold is just the special case $T_{\text{low}} = 0$ (or $T_{\text{high}} = 255$).

Where should T go? The answer hides in the **gray-level histogram**. If the object and the background are each uniform in gray value and clearly contrasted, the histogram shows two peaks — one for the foreground, one for the background — separated by a valley. This is the **bimodal histogram**. Place T at the bottom of the valley and the two populations are cut apart cleanly. The synthetic experimental scene of this chapter is designed exactly this way: on a 480×360 image, a bright background of gray value 160 carries 48 dark squares of 20×20 pixels (gray value 60, resembling a backlit dot calibration target), with additive Gaussian noise of standard deviation $\sigma = 6$

(fixed seed, reproducible); the ground-truth foreground totals 19200 pixels. Figure 7.2a is the uniformly illuminated version; Figure 7.2b is the same scene with a horizontal illumination gradient superimposed — a gain of $\times 0.6$ at the left end and $\times 1.2$ at the right, simulating uneven one-sided lighting.



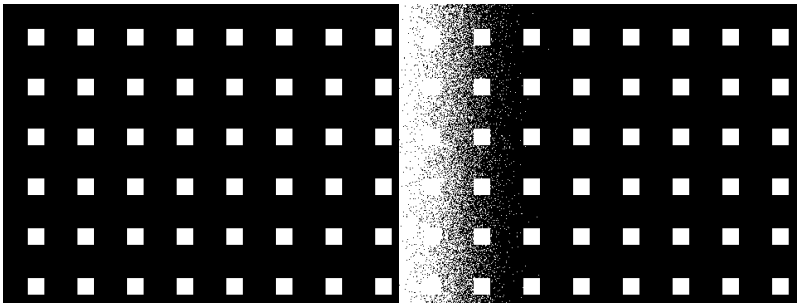
(a) Uniform-illumination scene (b) Horizontal-gradient scene

Figure 7.2: The synthetic test scenes. (a) Uniform illumination: background 160, squares 60, with additive Gaussian noise of $\sigma = 6$; (b) the same scene multiplied by a horizontal illumination gain ramping from 0.6 (left) to 1.2 (right) — dark on the left, bright on the right.

A fixed threshold of $T = 110$ (band $[0, 110]$) performs flawlessly on the uniform scene: out of 172800 pixels, **0 are misclassified** (Figure 7.3a). But move to the gradient scene and the very same T goes bankrupt at once (Figure 7.3b): at the left end the background gray value is pressed down by the gain to $160 \times 0.6 \approx 96$, below 110, so roughly the leftmost 70 columns of background fall wholesale into the threshold band — 23728 pixels misclassified (13.73%), every one of them a false positive in which background is mistaken for foreground.

T should sit in the middle of the valley, not at the foot of one of the peaks. With noise of $\sigma = 6$, each population spreads over roughly $\pm 3\sigma = \pm 18$ gray levels; place T too close to one peak and that population's "long tail" will spill across it. Between 60 and 160, choosing $T = 110$ leaves 50 levels of margin to each peak — ample headroom.

This controlled comparison spells out the precondition for fixed thresholding: **stable illumination and controlled contrast**. These two conditions are not granted by the algorithm — they are granted by the lighting design of Chapter 4: backlit silhouettes, narrow-band filters against ambient light, flat-field calibration — all of it exists to create the conditions under which "one T serves throughout". When the conditions hold, a fixed threshold is the fastest and most predictable solution; when they do not, even the most painstakingly hand-tuned T is



(a) Fixed $T = 110$, uniform scene (b) Fixed $T = 110$, gradient scene

Figure 7.3: The triumph and failure of a fixed threshold. (a) Uniform scene: all 48 squares are extracted cleanly, 0 misclassified; (b) gradient scene: the background gray value at the left end drops below 110 and the whole region is mistaken for foreground — 23728 pixels misclassified (13.73%, all false alarms).

merely a down payment on the next illumination fluctuation.

7.2 Otsu's Automatic Threshold

Choosing T by hand requires an engineer staring at the histogram and tuning. **Otsu's method** automates that step: it sweeps every candidate threshold and picks the one that "separates the two populations the farthest". Let p_i be the normalized histogram at gray level i. A threshold T splits the pixels into two classes, whose proportions and means are

$$\omega_0 = \sum_{i=0}^{T} p_i, \quad \omega_1 = \sum_{i=T+1}^{255} p_i, \quad \mu_0 = \frac{1}{\omega_0}\sum_{i=0}^{T} i\,p_i, \quad \mu_1 = \frac{1}{\omega_1}\sum_{i=T+1}^{255} i\,p_i.$$

Otsu's criterion is to maximize the **between-class variance**:

$$\sigma_B^2(T) = \omega_0\,\omega_1\,(\mu_0 - \mu_1)^2, \qquad T^* = \arg\max_T \sigma_B^2(T).$$

Intuitively, $(\mu_0 - \mu_1)^2$ rewards "the two class centers being far apart", while $\omega_0\omega_1$ penalizes lopsided splits where one class is large and the other tiny. By the variance decomposition of Chapter 2, the total variance $\sigma^2 = \sigma_W^2 + \sigma_B^2$ (within-class plus between-class variance) is a constant independent of T, so maximizing the between-class variance is equivalent to minimizing the within-class variance — making each class as internally compact as possible.

On the uniform scene, Otsu automatically selects $T = 86$, again with 0 misclassifications. Figure 7.4 shows where it lands: at the bottom of the valley between the two peaks — just as good as the hand-tuned result, but requiring no human intervention at all. When the illumination brightens or dims as a whole (for example, as a light source ages), both peaks shift together, and the T Otsu selects follows them automatically — this is its genuine advantage over a fixed threshold.

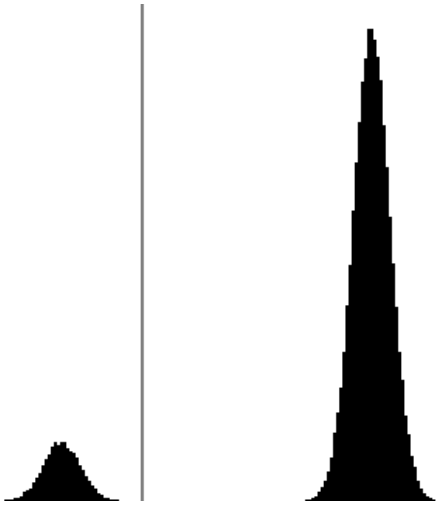


Figure 7.4: Gray-level histogram of the uniform scene. The small peak on the left is the foreground squares (centered near 60, 19200 pixels); the large peak on the right is the background (centered near 160). The vertical line marks Otsu's automatically selected threshold $T = 86$, landing exactly in the valley between the two peaks.

But look at the result on the gradient scene (Figure 7.5): Otsu

selects $T = 129$, misclassifying 52475 pixels (30.37%) — **more than twice as bad as the fixed threshold's 13.73%**. The reason is not hard to see: the horizontal gradient smears the background gray value from a single 160 into a broad ramp spanning 96 to 192; the right-hand peak of the histogram is flattened into one continuous plateau, and the "two peaks with a valley" structure no longer exists. Otsu still faithfully maximizes the between-class variance, but on the distorted histogram, $T^* = 129$ falls inside the widened background mode — roughly the left third of the background has gray values below 129, and the whole stretch turns into false alarms.

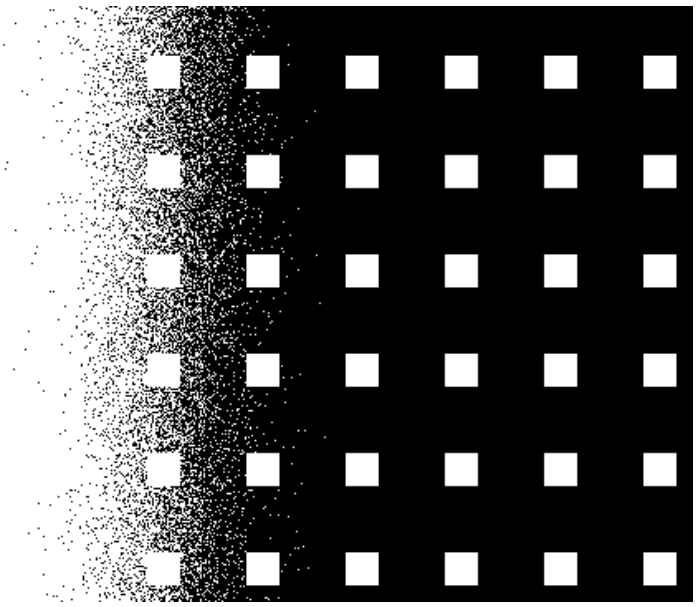


Figure 7.5: Otsu's failure on the gradient scene: it automatically selects $T = 129$, a large swath of background on the left side of the image is mistaken for foreground, and 52475 pixels are misclassified (30.37%) — worse than the fixed threshold.

This is the most important teaching point of the chapter: **Otsu's implicit assumption is that the histogram is approximately bimodal**. When the assumption holds, it is a tuning-free workhorse; when the assumption collapses, not only does it fail to save the day — its very appearance of being "automatic" makes it more deceptive. Engineers often

"Automatic" does not mean "adaptive". What Otsu automates is the act of choosing T, not adaptation to spatial variation — it still outputs a single global T. The histogram is a statistic that erases spatial information entirely: what an illumination gradient destroys is precisely the premise that "pixels of the same class share the same gray value", and no histogram algorithm, however clever, can repair that.

realize only after a batch of misjudgments on the line that the automatically selected threshold had been sitting in the wrong place from day one.

7.3 Adaptive Thresholding

The essence of an illumination gradient is that the threshold ought to vary with position. **Adaptive thresholding** answers this head-on — instead of a global statistic, it opens a local $K \times K$ window around every pixel and sets the threshold on the spot from the statistics inside the window. The most common form is "local mean minus an offset":

$$T[n, m] = \mu_K[n, m] - \beta,$$

where μ_K is the local mean and β is a fixed offset. A pixel is judged (dark) foreground if it is darker than the average level of its own neighborhood by more than β. Within the scale of the window, the illumination gradient is approximately constant: it raises the pixel value and the local mean by the same amount, so the subtraction cancels it automatically — exactly the "spatial adaptivity" that no global method can offer. The offset β, in turn, suppresses noise in flat regions: without β, about half the pixels on a uniform background naturally fall below the local mean, and a snowstorm of speckles erupts.

On the gradient scene, the result of local adaptive thresholding ($K = 81$, $\beta = 20$) is shown in Figure 7.6: only **41 pixels misclassified (0.02%)** — three orders of magnitude below the fixed threshold's 23728 and Otsu's 52475, with the squares at the darkest left end and the brightest right end extracted on equal terms.

The price is two new parameters. The lower bound of β is set by the noise (it should exceed roughly 3σ; here $\sigma = 6$, so $\beta = 20$), and its upper bound by the object contrast. The rule for K is in the margin note: **the window must be larger than the object size**, or the object's interior gets hollowed out; at the same time the window must be smaller than the

The window must be clearly larger than the object — a hard rule we established empirically: in this experiment, shrinking the kernel from 81 to 41 makes the interiors of the squares start to get "hollowed out". When most of the window lies inside a 20 px square, the local mean itself approaches the square's gray value, the center pixel is no longer darker than $\mu_K - \beta$, and the square's interior is judged background, leaving only a ring of outline. With 20 px objects, only a kernel of 81 is stable (the SDK's local mean is center-weighted, so the effective window is smaller than the nominal size).

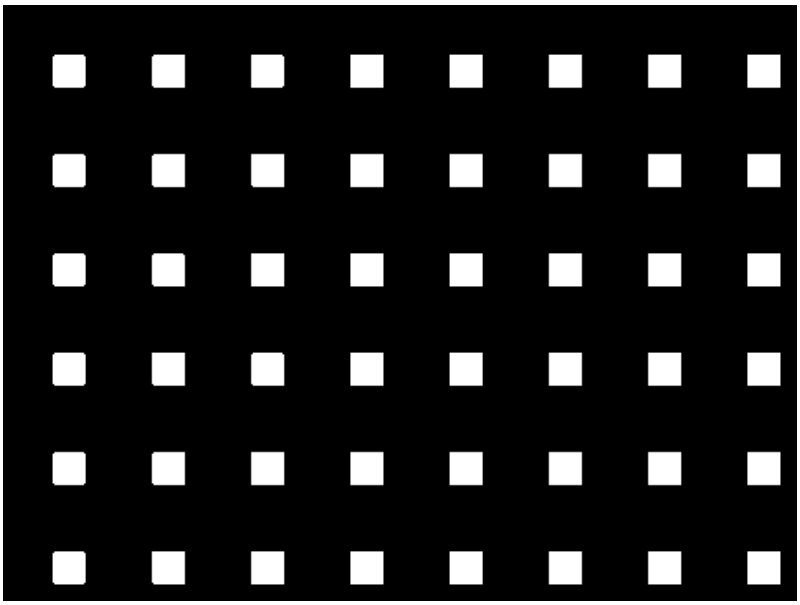


Figure 7.6: Local adaptive thresholding ($K = 81$, $\beta = 20$) on the gradient scene: only 41 pixels misclassified (0.02%); the illumination gradient is cancelled automatically by the local statistics.

scale of the illumination variation, or the method degenerates back into global thresholding. Leave margin on both ends.

Finally, let us set this chapter side by side with the route taken in Chapter 4. Faced with uneven illumination, we now have two equivalent remedies: **route one** — first apply shading correction to flatten the background, then use a global threshold (the approach of Chapter 4); **route two** — skip the correction and apply local adaptive thresholding directly (the approach of this chapter). Their mathematical core is in fact the same — both "subtract a local estimate of the background"; the only difference is whether that step happens in the gray-value domain or in the decision domain. The engineering trade-off: if the corrected gray image will go on to serve other algorithms such as measurement or OCR, choose route one — one correction benefits the whole pipeline; if all you need is this one binary image, route two saves a processing stage and a round trip of an intermediate image through memory.

The four experiments are summarized below (172800 pixels total, ground-truth foreground 19200):

Method	Scene	Threshold	Misclassified (pixels)	Error rate
Fixed $T = 110$	Uniform	110 (manual)	0	0%
Fixed $T = 110$	Gradient	110 (manual)	23728	13.73%
Otsu	Uniform	86 (automatic)	0	0%
Otsu	Gradient	129 (automatic)	52475	30.37%
Adaptive $K = 81$	Gradient	per-pixel	41	0.02%

7.4 Binarizing Color Images

The thresholding idea extends directly to color images: replace the single-channel band $[T_{\text{low}}, T_{\text{high}}]$ with one band per channel, and take the intersection of the three bands as the foreground — in RGB space this amounts to enclosing the target color in an axis-aligned box. The limitation of RGB is that all three channels are entangled with brightness; once the illumination changes, the box no longer contains the target. In practice the more robust approach is to convert to HSV space first: select the color with the hue channel, exclude gray and white

regions with the saturation channel, and leave the brightness channel loose or unconstrained — color identity is thereby decoupled from illumination intensity, far more robust to lighting fluctuations than an RGB box. SciVision's color binarization interface (SDK manual section 5.3) works in exactly this per-channel band-threshold fashion. The companion project of this chapter focuses on grayscale experiments and does not include a quantitative comparison for color binarization; only the principle is given here.

7.5 SciVision Implementation

The thresholding algorithms of this chapter are provided by the `SCIMV::SciSvThreshold` class. The fixed (manual) threshold call matches the companion project:

```
SCIMV::SciSvThreshold th;
SciImage dst;
SciROI roi;
SciPoint tl(0, 0), br(W, H);   // GenRect1 treats the bottom-right corner as exclusive: pass (W
roi.GenRect1(tl, br);

// Band threshold [0,110]: pixels whose gray values fall inside the band become white (255) in
long rc = th.ManualThreshold(img, roi, 0, 110, SCI_THRESHOLD_TYPE_WHITE, 0, &dst);
```

In `ManualThreshold(src, roi, lower, upper, lightOrDark, checked, dst)`, `lower`/`upper` are the bounds of the threshold band. The parameter name `lightOrDark` strongly suggests that it selects "extract bright objects or dark objects" — **our tests show it does not: it controls the output coloring, not the object selection**. Passing 0 (`SCI_THRESHOLD_TYPE_WHITE`) makes in-band pixels white (255) in the output. When extracting dark objects with the band $[0, 110]$, you must pass 0 so that the objects come out as 255, consistent with the output convention of `AutoThreshold` (foreground is white); pass the wrong value and the binary image is inverted, and the connected-component statistics of Chapter 23 downstream will count the background as the object.

Automatic thresholding (Otsu and local adaptive) shares a single interface:

```
int t = -1;
// autoThresholdType=1: Otsu; the automatically selected threshold is returned via t (kernelSi
rc = th.AutoThreshold(img, roi, SCI_THRESHOLD_TYPE_BLACK, 1, 3, 5, 0, &t, &dst);

// autoThresholdType=6: local adaptive; kernelSize=81 is the window side length, belta=20 is t
rc = th.AutoThreshold(img, roi, SCI_THRESHOLD_TYPE_BLACK, 6, 81, 20, 0, &t, &dst);
```

Parameter	Meaning	Value used here, with notes
`lightOrDark`	Object polarity	`SCI_THRESHOLD_TYPE_BLACK` (extract dark objects)
`autoThresholdType`	Algorithm selection	1 = Otsu; 6 = local adaptive
`kernelSize`	Local window side length K	81 (must be clearly larger than the 20 px objects; 41 hollows out the squares)
`belta`	Local offset β	20 (should exceed the noise spread of roughly 3σ)
`value`	Output: the automatically selected threshold	Under Otsu, returns the selected T (86 on the uniform scene, 129 on the gradient scene)

Two engineering details deserve a dedicated record. The first is the `lightOrDark` semantics described above. The second is that `SciROI::GenRect1` treats the **bottom-right corner as an exclusive endpoint**: a full-image ROI must be given (W, H). If, following the intuition of "the coordinates of the last pixel", you pass $(W-1, H-1)$, the last row and last column — $W+H-1 = 839$ pixels in total — are excluded from

processing and keep their original gray values: in the binary image this is an inconspicuous "leaky border", yet more than enough to derail any pixel-by-pixel verification. The complete project that generates all experimental images of this chapter lives in `code/thresholding/`.

Industry Case: Threshold Drift in Backlit Measurement

A backlit dimensional-measurement station originally used a fixed threshold to extract the workpiece silhouette. After several months of operation, the measured dimensions began to drift slowly and systematically — the investigation traced it to aging of the backlight's LEDs: brightness declined month by month, the contour cut by the fixed threshold expanded outward accordingly, and the dimensions were "measured larger" bit by bit. Switching to Otsu made the problem disappear: when the light intensity decays as a whole, both histogram peaks shift together, the automatic threshold follows, and the measurement stabilized. Then one day the line suddenly produced a batch of misjudgments: local contamination on the light guide plate dragged a long tail out of the background peak, the bimodal structure of the histogram was distorted, the threshold Otsu selected plunged into the background mode, and the contours of the entire batch were wrong — a live replay of this chapter's gradient experiment. The final solution was Otsu plus a **bimodality check**: before every segmentation, verify the distance between the two histogram peaks and the depth of the valley, and raise an alarm and halt judgment whenever the metrics go out of bounds. "Automatic" must be paired with "self-checking" before it can run unattended on a production line.

7.6 Summary

- **Thresholding is the first step from pixels to objects**: one gray-level threshold (or band) cuts the image into a binary foreground/background map. Its ceiling is set by the lighting — with stable illumination and controlled contrast, a fixed threshold is the fastest and most predictable (0 misclassified on the uniform scene); the

moment the illumination develops a gradient, the same threshold fails outright (13.73%).

- **Otsu selects the threshold automatically by maximizing the between-class variance** $\sigma_B^2(T) = \omega_0\omega_1(\mu_0 - \mu_1)^2$, and can follow global brightening or dimming of the illumination; but its implicit assumption is an approximately bimodal histogram. When that assumption collapses it can be worse than hand tuning (30.37% versus 13.73% on the gradient scene) — "automatic" does not mean "adaptive".
- **Adaptive thresholding** $T[n, m] = \mu_K[n, m] - \beta$ **cancels illumination gradients with local statistics** (misclassification drops to 0.02%). The window K must be clearly larger than the object size, or the object's interior gets hollowed out (kernel 41 fails; 81 is stable); β must exceed the noise spread.
- **There are two equivalent remedies for uneven illumination**: shading correction followed by a global threshold (Chapter 4), or local adaptive thresholding directly (this chapter). Choose the former when the intermediate gray image has other uses; choose the latter when all you need is the binary image.
- **The binary image is a midpoint, not an endpoint**: the remaining isolated specks and ragged edges are handed to the morphological cleanup of Chapter 8, and the aggregation of foreground pixels into objects and their measurement is the subject of Chapter 23.

The Otsu method used in this chapter comes from its 1979 original paper (Otsu 1979); for a systematic comparison and quantitative evaluation of the many thresholding methods, see the survey by Sezgin and Sankur (Sezgin and Sankur 2004); and for the place of thresholding within the overall digital-image-processing framework, see the textbook by Gonzalez and Woods (Gonzalez and Woods 2018). For a systematic treatment of thresholding and its connection to region-extraction algorithms, see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

8 Morphology

The output of thresholding (Chapter 7) is never a clean binary image. However well the threshold is chosen, the foreground is always left with a few pepper-like small holes, burrs hanging off the edges, two targets that should be separate glued together by a thin bridge, one target that should be whole split into pieces by a thin gap, and isolated noise dots scattered across the background. To the human eye these blemishes are trivial; to a program they are a disaster: the downstream blob analysis (Chapter 23) will count every noise dot as a target, count two merged targets as one, and count one fractured target as three. **Mathematical morphology is the wrench that repairs binary images** — it slides a small template across the image and, following the rules of set operations, deletes or fills in pixels, trimming the segmentation result into shapes the downstream algorithms can digest directly.

Throughout this chapter a single 480×360 synthetic binary scene runs through all the experiments (Figure 8.1). Every blemish in the scene is placed deliberately: a solid rectangle with 1–2 px pepper holes inside and 1–2 px burrs on its edges; two blobs merged by a 2 px thin bridge; a rectangle split into three pieces by 1 px and 2 px thin gaps; a 3 px-thick thin strip; plus 10 isolated noise dots (six 1×1 and four 2×2). All coordinates are hard-coded with no random numbers, so the statistics are identical on every run: 47426 foreground pixels in total, forming 16 blobs under 8-connectivity.

8.1 Structuring Elements, Erosion, and Dilation

Every morphological operation revolves around one protagonist: the **structuring element**. It is a small shape template

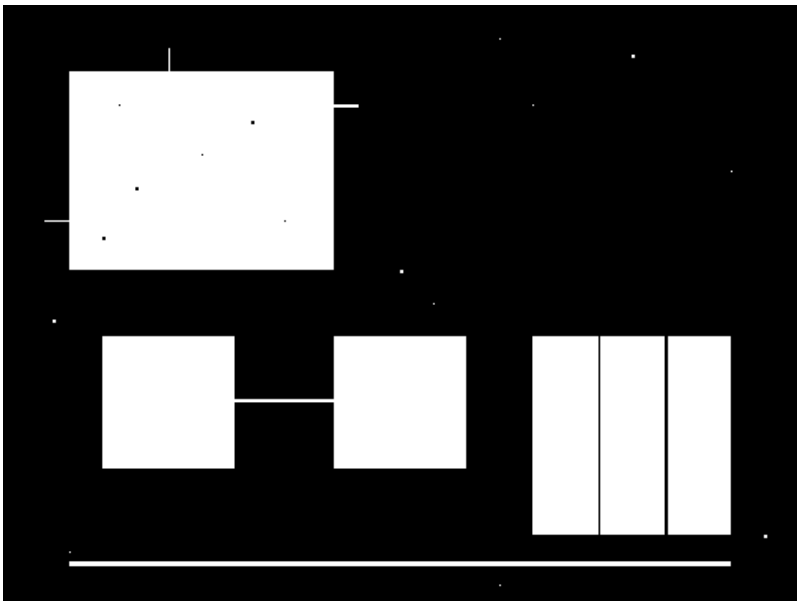


Figure 8.1: The synthetic binary scene used throughout this chapter: a rectangle with pepper holes and burrs, a double blob merged by a 2 px thin bridge, a rectangle split by 1 px/2 px thin gaps, a 3 px thin strip, and 10 isolated noise dots. Foreground 47426 px, 16 blobs under 8-connectivity.

with an **anchor** — most commonly a 3×3 or 5×5 rectangle with the anchor at the center. During the operation the anchor scans the image pixel by pixel, and the neighborhood covered by the structuring element decides that pixel's fate. The structuring element is to morphology what the convolution kernel is to linear filtering (Chapter 6): the template's shape and size determine the operation's entire character.

The two most basic operations form a pair of "antonyms." **Erosion** is defined in set terms as

$$A \ominus B = \{\, z \mid B_z \subseteq A \,\},$$

where A is the set of foreground pixels and B_z denotes the structuring element B translated so that its anchor sits at position z. In one sentence: **the anchor position stays foreground only if the structuring element fits entirely inside the foreground**. Anywhere that cannot accommodate the structuring element — thin bridges, burrs, isolated small dots, the outermost ring of every target — is stripped away. **Dilation** is the opposite:

$$A \oplus B = \{\, z \mid B_z \cap A \neq \emptyset \,\},$$

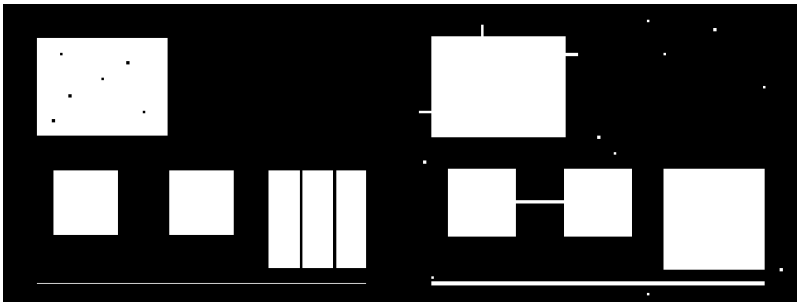
as long as the structuring element **touches the foreground in even one pixel**, the anchor position is set to foreground — the foreground grows outward by one ring, and thin gaps and small holes are filled by the foreground growing in from the opposite side.

For binary images there is an even more intuitive equivalent view: erosion is a neighborhood **min filter** — if the window contains even one background pixel (0), the output is 0; dilation is a neighborhood **max filter** — if the window contains even one foreground pixel (255), the output is 255. This view will come directly into play when Section 8.4 generalizes to grayscale morphology.

Applying each once to the test scene with a 3×3 rectangular structuring element gives Figure 8.2. After erosion the foreground drops from 47426 px to 44233 px, and the blob count

Erosion and dilation are each other's **dual (duality)**: $(A \ominus B)^c = A^c \oplus \check{B}$, i.e., eroding the foreground is equivalent to dilating the background. This is not mathematical decoration — it means that "deleting small foreground structures" and "filling small background gaps" are essentially the same operation viewed from the two sides, and the duality of opening and closing (next section) stems precisely from it.

plummets from 16 to 7: all 10 noise dots vanish, the burrs are stripped clean, and the 2 px thin bridge is severed — the merged double blob splits in two. The cost is equally visible to the naked eye: every target has slimmed down by one ring, the pepper holes are enlarged, and the thin gaps grow wider. Dilation does the opposite: the foreground swells to 50603 px, the pepper holes are filled, the 1 px and 2 px thin gaps both close up, but every target has fattened by one ring, and the noise dots, far from disappearing, have grown.



(a) 3×3 erosion (b) 3×3 dilation

Figure 8.2: Erosion and dilation (3×3 rectangular structuring element). (a) Erosion: noise dots and burrs are removed entirely and the thin bridge is severed (16→7 blobs), but the foreground shrinks to 44233 px and the holes and gaps are enlarged; (b) dilation: pepper holes are filled and thin gaps are closed, but the foreground swells to 50603 px and the noise dots grow as well.

Each operation solves only half of the problem, and each introduces a new one — **the size changes**. In measurement applications this is unacceptable: erode once, and the target's area and contour positions are systematically biased small; dilate once, and they are biased large. How can we enjoy the benefits of "deleting noise / filling gaps" while getting the size back? The answer is to chain the two together.

8.2 Opening and Closing

Opening is defined as erosion followed by dilation:

$$A \circ B = (A \ominus B) \oplus B,$$

while **closing** is dilation followed by erosion:

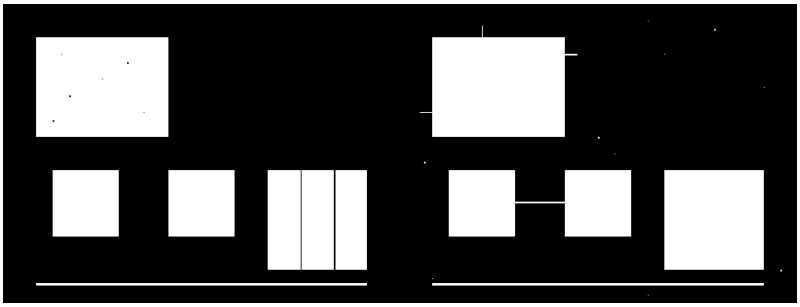
$$A \bullet B = (A \oplus B) \ominus B.$$

The design idea behind both is the same: the first step "kills or spares" the target structures, and the second step approximately restores the survivors' size. In opening, the erosion first deletes every small structure that cannot accommodate the structuring element — once gone, the subsequent dilation cannot bring them back; large targets, on the other hand, have merely slimmed by one ring, and the dilation puts that ring back. Closing acts dually on the background: the dilation first fills every small gap and hole that cannot accommodate the structuring element, and the erosion then shrinks the swollen targets back to their original shape.

The experimental results are shown in Figure 8.3, and the numbers demonstrate "approximate size preservation" with great clarity. After the 3×3 opening the foreground is 47225 px, a mere 0.4% decrease relative to the original 47426 px; after the 3×3 closing it is 47801 px, a mere 0.8% increase — compared with the roughly ±6.7% change of erosion/dilation alone, opening and closing reduce the size deviation by an order of magnitude. And the decrease and increase are exact to the pixel: the 201 px deleted by the opening equals precisely the combined area of the 10 noise dots (22 px), the three burrs (59 px), and the thin bridge (120 px) — the rectangle bodies did not lose a single pixel; the 375 px added by the closing equals precisely the combined area of the two thin gaps (360 px) and all the pepper holes (15 px).

Opening and closing are both **idempotent**: $(A \circ B) \circ B = A \circ B$, and likewise for closing. Doing it once and doing it a hundred times give exactly the same result — everything that should be deleted is deleted cleanly the first time, and the surviving structures already "accommodate the structuring element," so further applications are just the identity. This is in stark contrast to the cumulative "the more you filter, the blurrier it gets" behavior of smoothing filters.

The blob counts further reveal the **division of labor** between the two. After opening, 7 blobs remain: the 10 noise dots are gone, the thin bridge is severed (the merged pair splits in two), and the burrs are removed clean — but the pepper holes inside the rectangle are still there. **Opening does not fill holes**: it only deletes small protruding structures in the foreground and is powerless against small holes in the background. After



(a) 3×3 opening (b) 3×3 closing

Figure 8.3: Opening and closing (3×3 rectangular structuring element). (a) Opening: noise dots, burrs, and the thin bridge are removed entirely (16→7 blobs) while target size is nearly unchanged (−0.4%), but the pepper holes remain untouched; (b) closing: the 1 px and 2 px thin gaps are all sealed (the three pieces rejoin into one, 16→14 blobs) and the pepper holes are filled (+0.8%), but the noise dots and burrs are left exactly as they were.

closing, 14 blobs remain: the 1 px and 2 px gaps are both sealed, the three severed pieces rejoin into one, and the pepper holes are filled — but not one of the 10 noise dots is gone. **Closing removes neither burrs nor noise dots.** This is duality made visible: opening repairs the "excess" of the foreground, closing repairs the "excess" of the background, and neither can substitute for the other. In real engineering the two are often chained — open first, then close: sweep away the noise before sealing the broken gaps. The counts for all operations in this chapter are summarized in Table 8.1.

Table 8.1: Effect of each morphological operation on the test scene's foreground pixels and blob count

Operation	Foreground pixels	Relative to original	Blob count (8-connectivity)
Original scene	47426	—	16
Erosion 3×3	44233	−6.7%	7

Operation	Foreground pixels	Relative to original	Blob count (8-connectivity)
Dilation 3×3	50603	+6.7%	—
Opening 3×3	47225	−0.4%	7
Closing 3×3	47801	+0.8%	14
Opening 5×5	46025	−3.0%	6

8.3 Choosing the Structuring Element Size

Opening and closing hand the decision of "what to delete, what to keep" entirely to the size of the structuring element — whatever cannot accommodate it dies, whatever can survives. So is a bigger structuring element always safer? Swap the opening's structuring element from 3×3 to 5×5 and run again (Figure 8.4): the foreground drops to 46025 px and only 6 blobs remain. The one that went missing is not noise — it is the 3 px-thick thin strip. It accommodates a 3×3 structuring element but not a 5×5 one, so it was wiped out root and branch. If that "thin strip" were a lead wire or a genuine scratch on the product, the inspection system would have destroyed the evidence with its own hands at the preprocessing stage.

This yields the most important engineering rule of the chapter: **the structuring element must be larger than the noise to be deleted and smaller than the smallest structure to be kept**. In this example the largest noise is 2×2 and the smallest genuine structure is 3 px thick, so 3×3 lands exactly in the gap while 5×5 oversteps it. If the two bounds conflict — the noise grains are larger than the smallest target — then the single dimension of size can no longer separate them; do not force the structuring element, but switch to structuring elements of different shapes (an elongated structuring element for elongated targets) or leave the residual noise to downstream area-based screening.

This rule is isomorphic to the median-filter kernel-size rule of Chapter 6: the median kernel must be smaller than twice the width of the smallest structure to be preserved, or thin lines are wiped out together with the noise. This is no coincidence — opening and the median filter both belong to the family of nonlinear filters that "keep or discard by size ranking"; **neither knows "defect" from "noise" — they only know size**.

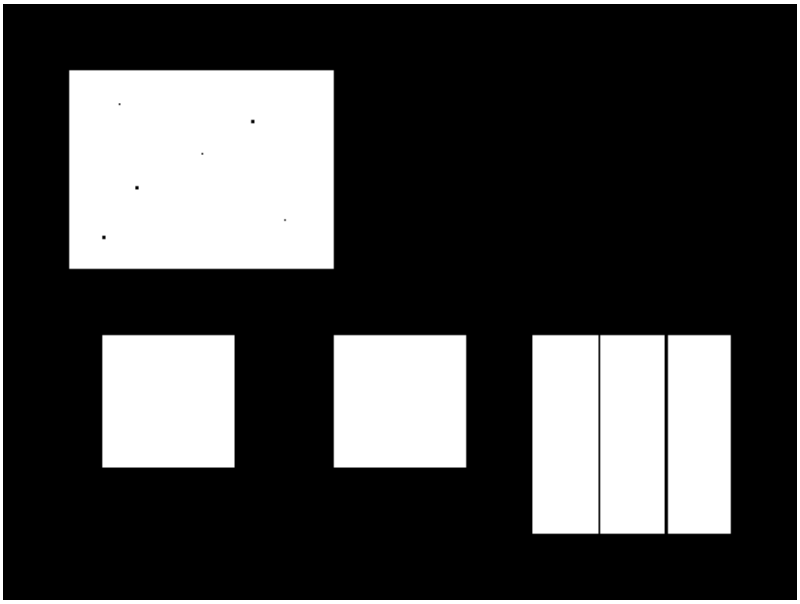


Figure 8.4: 5×5 opening: the noise is removed just as cleanly, but the 3 px thin strip is wiped out in its entirety as well (blob count 16→6) — once the structuring element is larger than the structures to be kept, deletion no longer distinguishes noise from target.

8.4 Grayscale Morphology and the Top-Hat

The min/max view mentioned in Section 8.1 lets morphology walk naturally out of the binary world: for grayscale images, **grayscale erosion** takes the **minimum** gray value within the structuring element's neighborhood, and **grayscale dilation** takes the **maximum**. Grayscale opening (min then max) shaves off every bright peak "too narrow to accommodate the structuring element," while leaving large-area background undulations almost untouched — in other words, **grayscale opening is an operation that keeps only the background**.

This immediately spawns an industrial power tool. The **white top-hat** transform is defined as the original image minus its grayscale opening:

$$T_b(f) = f - (f \circ b).$$

$f \circ b$ is "an estimate of the background with the small bright structures wiped out"; subtracting it from the original leaves exactly those small bright structures themselves — whether the background is flat, slanted, or curved, it is subtracted away clean. **The top-hat is naturally immune to the background brightness level**, which is exactly what shading correction in Chapter 4 tries to achieve: think of the top-hat as "small-target extraction with a built-in background estimate" — the background model needs no separate capture and no fitting; it is hidden inside the opening.

The experimental scene (Figure 8.5) is deliberately designed to defeat the global thresholding of Chapter 7: the background is a horizontal gradient from 30 to 160, on which 16 small bright spots (ranging from 2×2 to 5×5) are superimposed, each 70 gray levels brighter than its local background. The darkest spot, located on the left side of the image, peaks at only about 116, while the brightest background on the right is already 160 — **the spots are darker than the background, so no global threshold capable of separating the two exists**. After a white top-hat with an 11×11 structuring element (larger than the largest 5×5 spot), the gradient background is subtracted

(a) Grayscale scene (b) 11×11 white top-hat (c) Threshold 35 after top-hat

Figure 8.5: Extracting small bright spots under nonuniform illumination. (a) On a horizontal gradient background running from 30 to 160, 16 small bright spots of "background + 70" are superimposed; the darkest spot peaks at about 116, darker than the 160 background on the right side — any global threshold is doomed; (b) the 11×11 white top-hat subtracts the gradient background wholesale, leaving only the 16 bright spots; (c) binarization at threshold 35: exactly 16 blobs and 216 px, matching the ground-truth spot area to the pixel.

wholesale; binarizing at threshold 35 then extracts exactly 16 blobs totaling 216 px — in perfect agreement with the ground-truth area of the 16 spots ($4 \times (4 + 9 + 16 + 25) = 216$), not one pixel more, not one less. The top-hat's structuring-element size rule is the same as in the previous section, but flipped: **it must be larger than the targets to be extracted**, so that the opening wipes them out of the background estimate and they appear in the difference.

8.5 SciVision Implementation

The morphology functionality of the SciVision SDK is provided by two classes, `SciMorphStructure` (structuring element) and `SCIMV::SciSvMorphology` (operations). The invocation consistent with this chapter's experiments is as follows:

```
// Structuring element: CreateElement(rows, cols, anchorX, anchorY, shape, data)
SciMorphStructure se3, se11;
```

```
long rc = se3.CreateElement(3, 3, 1, 1, SCI_MS_RECT);    // 3×3 rectangle, anchor (1,1) centere
if (rc) { /* return codes must be checked */ }
rc = se11.CreateElement(11, 11, 5, 5, SCI_MS_RECT);      // 11×11 for the top-hat

SciROI roi;
SciPoint tl(0, 0), br(W, H);   // GenRect1's bottom-right is an exclusive endpoint: pass (W,H)
roi.GenRect1(tl, br);          // takes non-const references, so named variables are required

SCIMV::SciSvMorphology m;
SciImage dErode, dOpen, dTop;
rc = m.Erode(src, roi, se3, 1, &dErode);     // erosion, iteration = 1
rc = m.Open (src, roi, se3, 1, &dOpen);      // opening
rc = m.TopHat(gsrc, roi, se11, 1, &dTop);    // white top-hat
```

The first two parameters of `CreateElement` are the structuring element's row and column counts; `anchorX`/`anchorY` specify the anchor position (conventionally the center, e.g., (1,1) for 3×3 and (5,5) for 11×11); `shape` set to `SCI_MS_RECT` means a solid rectangle, and the final `int* data` parameter is used only for custom shapes — it can be omitted for rectangular kernels. The five interfaces `Erode/Dilate/Open/Close/TopHat` share the same signature: input image, ROI, structuring element, iteration count, and output pointer. The `iteration` parameter is the number of times the operation is repeated — for rectangular structuring elements, n iterations of a 3×3 erosion are equivalent to a single $(2n+1)\times(2n+1)$ erosion, so small-kernel iteration can substitute for a large kernel; but for opening and closing, idempotence means `iteration` greater than 1 may give the same result as 1, so it is normally fixed at 1.

One pitfall that must be reported honestly: the bottom-right corner of `SciROI::GenRect1` is an **exclusive endpoint** (the same semantics were already recorded in the SciVision section of Chapter 7). If you follow the "coordinate of the last pixel" intuition and pass $(W-1, H-1)$, the last row and last column are excluded from the ROI and left unprocessed — the morphology output is identically 0 along that border band, the top-hat difference $f-0$ degenerates to the original gray value, and the brighter side of the gradient background is left with a strip of high response strong enough to punch through the threshold.

This artifact is easily misread as an SDK defect of "not processing the outermost row/column"; in fact, once you pass (W, H) (or simply use an undefined ROI), the SDK processes the complete image and the artifact vanishes. All experiments in this chapter were run with a full-image ROI of (W, H).

Industry Case: Opening to the Rescue in Solder-Joint Inspection

At a PCB solder-joint inspection station, solder-paste splatter left large numbers of 1–3 px small bright grains on the board surface; after thresholding they all became foreground, and the blob count overstated the true number of solder joints several times over. Inserting a single 3×3 opening between segmentation and counting removed the splatter grains clean and the count returned to normal — a textbook application of opening. Three months later an engineer changed the structuring element to 7×7 on the grounds that it was "safer": the splatter was indeed removed even more thoroughly, but the line immediately began missing joints — the smallest-specification solder joint on that product was only 6 px in diameter, could not accommodate a 7×7 structuring element, and was wiped out together with the splatter. The post-mortem conclusion was written into the department standard: **the upper bound on the structuring element size must be derived backward from the minimum solder-joint size in the process documentation**, not forward from "how cleanly the noise is removed"; leaving residual splatter to downstream area-based screening is far safer than enlarging the structuring element.

8.6 Summary

- **Morphology is the repair step between thresholding and blob analysis**: erosion deletes pixels by the rule "the structuring element must fit entirely inside the foreground," dilation adds pixels by the rule "set the pixel whenever the structuring element touches the foreground"; the two are duals, equivalent to neighborhood min/max filters, but used alone each systematically

changes target size (about ±6.7% in this chapter's experiments).

- **Opening = erode then dilate: removes noise dots, burrs, and thin bridges but does not fill holes; closing = dilate then erode: seals thin gaps and fills small holes but does not remove noise dots.** Both keep the size deviation within ±1% (47426 → 47225 / 47801), and both are idempotent — once is enough.
- **The structuring-element size rule: larger than the noise to delete, smaller than the structures to keep** (3×3 spares the 3 px thin strip; 5×5 wipes it out). This is isomorphic to the median-filter kernel-size rule — nonlinear filters only know size, not semantics.
- **The white top-hat $f - (f \circ b)$ is small-bright-target extraction with a built-in background estimate**, immune to illumination gradients: in a gradient scene where any global threshold must fail, a single fixed threshold after the top-hat extracts all 16 bright spots exactly, with 216 px matching the ground truth. The structuring element must be larger than the targets to extract.
- **Engineering reminder**: the bottom-right corner of `GenRect1` is an exclusive endpoint — passing $(W - 1, H - 1)$ excludes the last row/column from processing, and the top-hat degenerates to the original gray value along that border band; a full-image ROI must pass (W, H).

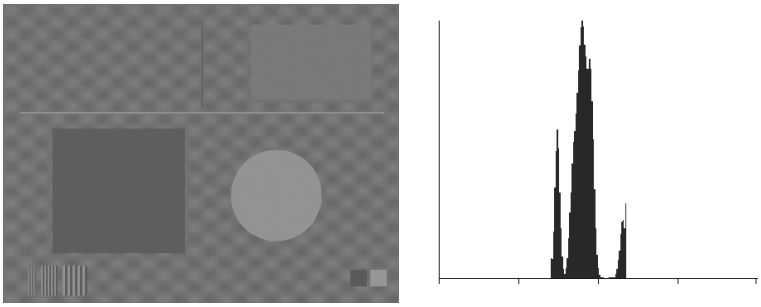
The theoretical origin of mathematical morphology is Serra's foundational monograph (Serra 1982), while Soille's book systematically covers grayscale morphology and its applications (Soille 2004); for a unified review of binary and grayscale morphological operators, see the classic paper by Haralick, Sternberg, and Zhuang (Haralick, Sternberg, and Zhuang 1987). For the complete theoretical edifice of morphology (advanced topics such as the hit-or-miss transform, geodesic reconstruction, and the watershed), see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

9 Gray-Level Transforms and Histograms

One of the most common complaints heard on a production line is "the image is too gray": the target and the background differ in gray level, yet to the eye they blur into one indistinct mass. Faced with such an image, the first instinct is to adjust the contrast — and "adjusting contrast" corresponds at the algorithmic level to the simplest class of image operations there is: the **point operation**. A point operation is defined by the property that the output gray level of a pixel depends only on the input gray level at the **same position**, i.e. $s = T(r)$, with no neighborhood involved whatsoever. This stands in sharp contrast to the spatial filtering of Chapter 6 — a filter looks at a neighborhood, a point operation looks only at itself. For an 8-bit image, every point operation reduces to one and the same implementation form: the **lookup table (LUT)** — precompute all 256 output values $T(0)$ through $T(255)$, store them in a table, and at processing time each pixel costs exactly one table lookup. The contrast stretching, gamma correction, and histogram equalization of this chapter are, at bottom, all answers to a single question: how should those 256 table entries be filled in.

Throughout this chapter we use one synthetic low-contrast scene (480×360) for all experiments: on a sinusoidal-texture background sit a flat plate of constant gray level, a dark rectangle, a bright circle, two thin 2 px lines, and a set of resolution stripes, plus two anchor blocks that pin the gray-level range precisely to 90 and 150 — the gray values of the entire image occupy only $[90, 150]$, using less than a quarter of the 8-bit dynamic range. Figure 9.1 shows the scene and its histogram.

The engineering value of the LUT is that it unifies "an arbitrarily complex gray-level mapping" into one $O(1)$ table lookup per pixel: first evaluate $T(r)$ once for each of the 256 gray levels, then sweep the image once. Whether T is piecewise linear, a power function, or a CDF, the running cost is exactly the same.



(a) Low-contrast scene (b) Histogram before processing

Figure 9.1: The low-contrast test scene and its histogram. (a) All gray values are compressed into $[90, 150]$; targets and background are hard to tell apart by eye; (b) the histogram confirms it: all the counts are squeezed into a narrow band in the middle of the axis, with broad stretches of completely empty gray levels on both sides.

9.1 The Histogram: The Image's Gray-Level Ledger

The **histogram** is a statistic of the image's gray-level distribution: the count $h(k)$ of the k-th bin equals the number of pixels whose gray value is k. Dividing the counts by the total number of pixels N gives the normalized histogram $p(k) = h(k)/N$ — which is precisely a concrete instance of the "empirical distribution" of Chapter 2: if "the gray value of a randomly drawn pixel" is viewed as a random variable, $p(k)$ is the empirical estimate of that variable's distribution. The histogram discards all spatial information (shuffle the image's pixels and the histogram is unchanged), yet it condenses everything about exposure and contrast into a single curve.

Reading a histogram is a basic skill. Counts piled up against 0 at the left end mean underexposure (the shadows are clipped); piled up against 255 at the right end, overexposure (the highlights are saturated, and no software can ever recover the clipped information — that has to be fixed back at the illumination design of Chapter 4); whereas counts squeezed into a narrow mid-range band, as in Figure 9.1b, are the

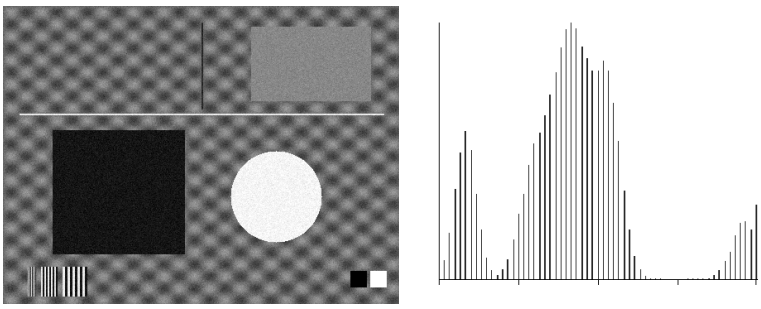
signature of **low contrast**: the exposure is fine, but the scene itself has little reflectance contrast or lens flare has lifted the shadows. For this chapter's scene, the SDK statistics give: mean 114.78, gray-level standard deviation 12.55, range exactly $[90, 150]$. A standard deviation of only 12.55 — for an 8-bit image, that is the quantitative way of saying "everything is one gray mush."

9.2 Linear Stretching

Since the gray values occupy only $[r_{\min}, r_{\max}] = [90, 150]$, the most direct fix is to stretch this interval linearly to fill $[0, 255]$:

$$s = (r - r_{\min}) \cdot \frac{255}{r_{\max} - r_{\min}}.$$

This is a straight line of slope $255/60 = 4.25$ — every pixel's gray value (noise included) is amplified by a factor of 4.25. Figure 9.2 shows the stretched result: the texture, the thin lines, and the stripe group are all clearly distinguishable; statistically the mean is 105.30, the standard deviation leaps from 12.55 to 53.34, and the range fills $[0, 255]$.



(a) Linear stretching result (b) Histogram after stretching

Figure 9.2: Linear stretching maps $[90, 150]$ onto $[0, 255]$. (a) The contrast improves dramatically and every structure is clearly visible; (b) the histogram is pulled apart into a **comb** — regular gaps are left between the bars.

The histogram after stretching (Figure 9.2b) exhibits the textbook **comb artifact**: the input contains only 61 discrete gray levels, and an integer LUT maps them to at most 61 output levels within $[0, 255]$ — the remaining nearly 200 bins are doomed to stay empty. Stretching **creates no new information whatsoever**; it merely spreads the existing 61 gray levels out along the axis, and the number of distinguishable gray levels does not increase by even one. This is also a useful diagnostic: when you see a comb-shaped histogram, you can be fairly confident the image has been digitally stretched.

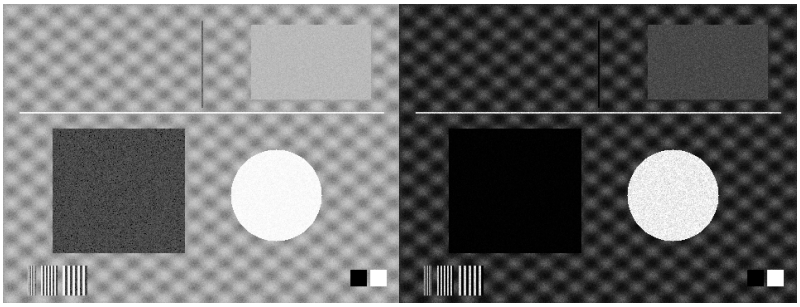
In practice, stretching directly from the image's global minimum and maximum is not robust — a single bad pixel can drag $r_{\min}$ or $r_{\max}$ to an extreme value and defeat the stretch. The standard engineering remedy is **percentile-clipped stretching**: discard the darkest and brightest $p\%$ (e.g. 1%) of pixels before taking the interval endpoints, buying immunity to outliers at the cost of sacrificing a tiny fraction of pixels.

9.3 Gamma Correction

Linear stretching treats all gray levels alike; to **selectively** brighten the shadows or darken the highlights, a nonlinear mapping is needed. The most common one is the power-law **gamma correction**:

$$s = 255 \cdot \left(\frac{r}{255}\right)^{\gamma}.$$

The intuition behind the curve is simple: for $\gamma < 1$ the curve bows upward — the slope in the shadows exceeds 1 (they are spread apart and brightened) while the slope in the highlights is below 1 (they are compressed); for $\gamma > 1$ it is the reverse — shadows compressed, highlights expanded. The endpoints $0 \mapsto 0$ and $255 \mapsto 255$ never move; only the allocation of the midtones changes. Probing the SDK's convention with a 0–255 gray ramp: at $\gamma = 0.5$, $64 \to 128$, $128 \to 181$, $192 \to 221$ — the output is always greater than the input, consistent with the formula above (input normalized first, then raised to the power).



(a) $\gamma = 0.5$ (brighten shadows) (b) $\gamma = 2.0$ (darken shadows)

Figure 9.3: Two gammas applied to the stretched image (Figure 9.2a). (a) $\gamma = 0.5$ brightens the image overall, and detail inside the dark rectangle emerges; (b) $\gamma = 2.0$ darkens the image overall — the dark rectangle goes nearly pitch black, while the tonal gradations in the bright regions actually open up.

Measured on the stretched image (Figure 9.3): the interior of the dark rectangle (mean 21.4 after stretching) is lifted to 71.8 by $\gamma = 0.5$ and crushed to a mere 2.3 by $\gamma = 2.0$; the interior of the bright circle (245.6) rises only slightly to 250.0 under $\gamma = 0.5$ and dips slightly to 236.8 under $\gamma = 2.0$. The numbers confirm the shape of the curve: nearly all of gamma's "muscle" is spent on the side far from the endpoints — $\gamma < 1$ moves the shadows dramatically while barely touching the highlights, and $\gamma > 1$ does the opposite.

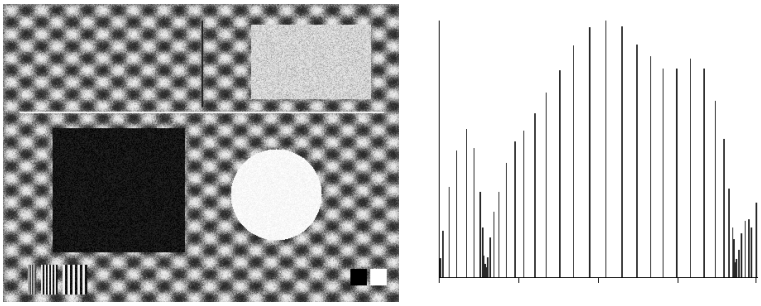
9.4 Histogram Equalization

Stretching and gamma both require parameters to be chosen by hand; **histogram equalization** lets the image decide its own mapping — the goal is to make the output histogram as flat as possible. The derivation takes just one step: to "flatten" the distribution, densely populated gray ranges should be spread apart and sparse ones compressed, and the **cumulative distribution function (CDF)** has exactly this property built in. Define $\mathrm{CDF}(r) = \sum_{k \le r} p(k)$ and take the mapping

The word "gamma" comes from the display chain: a traditional monitor's luminance response is approximately $V^{2.2}$, so consumer images (sRGB) are pre-encoded with the inverse $\gamma \approx 1/2.2$ at storage time. Industrial cameras, by contrast, output **linear** gray values by default — pixel value proportional to luminous flux, which is exactly what measurement algorithms want. Introducing gamma into a measurement pipeline means destroying the linear photometric relationship and should be done only with great caution; its legitimate uses are display and human visual inspection.

$$s = T(r) = 255 \cdot \mathrm{CDF}(r),$$

then T is monotonically nondecreasing (it never reverses the gray-level order), and its local slope is proportional to the histogram density $p(r)$ at that point — where pixels crowd together, the slope is steepest and the values are pulled apart the most. Equalizing this chapter's scene gives the result in Figure 9.4: the standard deviation rises further to 74.28 (mean 131.30), a contrast even beyond linear stretching.



(a) Equalization result (b) Histogram after equalization

Figure 9.4: Histogram equalization. (a) Contrast is maximized, but the flat plate and the background show pronounced graininess; (b) the histogram spreads across the entire gray axis (a discrete image cannot be made strictly flat; the bars are sparse and uneven in height).

But look closely at the "flat plate" in the upper right of Figure 9.4a: in the original it was a constant gray level of 122 plus $\sigma = 2$ noise, and now it is covered in grain. Quantitatively measuring the standard deviation inside this plate yields the most important set of numbers in this chapter: 2.02 in the original, 8.46 after linear stretching (a factor of 4.19 — almost exactly the LUT's constant gain of 255/60), but **18.02 after equalization — an amplification of 8.93×**, more than double that of linear stretching. The reason lies precisely in the slope of the CDF: the flat plate's several thousand pixels are all crammed into two or three bins around 122, where $p(r)$ is extremely high, so the local slope of T is at its steepest — **equalization amplifies gray-level differences (noise included) most fiercely**

exactly where pixels are most densely packed, and flat regions are precisely where pixels pack most densely.

This settles equalization's place in engineering: it is a tool for "showing images to people" — well suited to visual inspection, report figures, and manual-review interfaces; but it should be used with great caution in measurement and defect-detection pipelines. The automatic thresholding methods of Chapter 7 rest on the statistical premise that "foreground and background each have concentrated gray values," and equalization systematically destroys exactly that premise: once the noise in flat regions has been amplified nearly 9-fold, a previously clean background sprouts a host of spurious responses that cross the threshold.

Equalization's local gain $T'(r) \propto p(r)$ also explains a seemingly paradoxical fact: its boost to **genuine structural contrast** and its amplification of **flat-region noise** come from one and the same mechanism — you cannot have the former without the latter. Adaptive variants (such as CLAHE's clip limit) mitigate this conflict precisely by capping the slope at densely populated bins.

9.5 SciVision Implementation

This chapter's experiments involve three classes: histogram statistics `SciSvHistogram`, stretching and equalization `SciSvHistEqualize`, and gamma `SciSvBrightness`.

```
SCIMV::SciSvHistogram histOp;
SciROI roi;                          // default-constructed = UNDEF ROI, which the SDK takes to me
SciVarArray histData, histRange;
double avg, stdValue; int mn, mx, med, norm, total;
// lower=0, upper=255 statistics range; type=4 custom bin count; nBins=256, i.e. one bin per gr
histOp.CaculateHist(img, roi, 0, 255, 4, 256, &histData, &histRange,
                    &avg, &stdValue, &mn, &mx, &med, &norm, &total);

SCIMV::SciSvHistEqualize eq;
SciImage stretched, equalized;
// method=1 clipped linear stretching; lowPct/highPct=0 means stretch [minGray,maxGray]=[90,150
eq.LineStretchHist(scene, roi, 1, 0.0, 0.0, 90, 150, &stretched);
// rangeValue=255 full-range equalization; minStrength=0 / maxStrength=255 is the default stren
eq.EqualizeHist(scene, roi, 255, 0, 255, &equalized);

SCIMV::SciSvBrightness bright;
SciImage dst;
bright.AdjustGamma(img, roi, 0.5f, &dst);   // s = 255*(r/255)^gamma
```

A single call to `CaculateHist` returns the histogram array together with a set of statistics, but there is one pitfall you must know about: **the output parameter `stdValue` is the standard deviation of the 256 bin counts, not of the pixel gray values** — for this chapter's original image it returns 157.61, which already exceeds 127.5, the theoretical upper bound of an 8-bit gray-level standard deviation, so it is clearly not a gray-value statistic. The values 12.55/53.34/74.28 quoted in the text are all gray-level standard deviations computed from the histogram ourselves. `LineStretchHist` with `method=0` (automatic stretching from the global min-max) was found to apply no transform at all to this image — a suspected SDK defect — so we use `method=1` and specify `[minGray, maxGray]` explicitly. Also, do not construct the full-image ROI as `GenRect1((0,0),(W-1,H-1))` — that rectangle is treated as a half-open interval, and the last row and column, 839 pixels in total, are excluded from the statistics; simply use a default-constructed UNDEF ROI to denote the whole image. One final detail: a pixel-by-pixel comparison of the SDK's stretch against a hand-written LUT following the formula shows a maximum difference of just 1 (from truncation-versus-rounding differences in integer conversion), confirming that `LineStretchHist` is internally exactly a lookup-table point operation.

The complete runnable project is located at `code/gray_transforms_histograms/`, with fixed random seeds to ensure reproducibility.

Industry Case: Two Ways to Fix "the Image Is Too Dark"

A production line reported that its inspection images were dark and low in contrast. The software team's first "fix" was histogram equalization — the picture instantly became crisp and good-looking. Two weeks later the defect false-alarm rate had doubled: the sensor noise in the flat background regions had been amplified several-fold by the equalization (the 2.02→18.02 of this chapter replayed on the factory floor), and swarms of noise pixels crossed the defect-decision threshold and were reported as phantom defects. The correct fix, established in the post-mortem, came in two steps. Step one: go back to the hardware chain — check the exposure time, the aperture, and the lens; bring the analog gain down; and obtain, at the source, an image with sufficient gray-level range and controlled noise (see

Chapter 4 and Chapter 1). Step two: in software, apply only a parameter-controlled percentile-clipped linear stretch, and feed the post-stretch noise level back into the threshold tuning. The lesson fits in one sentence: **looking good and measuring well are two different things** — the high-contrast image the human eye likes is not necessarily the low-noise image the algorithm needs.

9.6 Summary

- **Point operation = lookup table.** Any operation whose output depends only on the input gray value at the same position is a point operation; on 8-bit images every one of them can be implemented as a 256-entry LUT. Stretching, gamma, and equalization are merely three different strategies for filling that table.
- **The histogram is the empirical distribution of gray values**: it throws away spatial information yet retains all the evidence about exposure and contrast — piled against the left end means underexposure, against the right end means overexposure (the information is unrecoverable), squeezed into a narrow mid-range band means low contrast.
- **Linear stretching creates no information**: spreading 61 gray levels across 256 leaves the number of distinguishable levels unchanged and leaves comb-shaped gaps in the histogram; the endpoints should be clipped percentiles rather than the global min-max, to resist outliers.
- **Gamma correction redistributes the midtones nonlinearly**: $\gamma < 1$ lifts the shadows and compresses the highlights, $\gamma > 1$ the reverse, with the endpoints fixed; industrial cameras output linear gray values by nature, so introducing gamma into a measurement pipeline demands caution.
- **Equalization fills the table from the CDF, with local gain proportional to histogram density** — flat regions are where pixels are densest and where noise is amplified most fiercely (8.93× measured in this chapter, far beyond linear stretching's 4.19×). It is an enhancement

tool for human viewing; before letting it into a measurement/inspection pipeline, first assess the damage to the statistical premises of Chapter 7.

The standard treatment of gray-level transforms and histogram processing (including global histogram equalization) is the textbook by Gonzalez and Woods (Gonzalez and Woods 2018); to counter the very drawback of global equalization amplifying noise in flat regions, the adaptive histogram equalization and its clipped variant (the precursor of CLAHE) proposed by Pizer et al. is the classic remedy (Pizer et al. 1987). For a systematic treatment of gray-level transforms and histogram processing in industrial pipelines (including robust stretching and polynomial gray-level correction), see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

10 Geometric Transforms

Workpieces on a production line never show up in the field of view at the same position and the same angle every time: vibratory bowl feeders deliver parts with scattered poses, pallet positioning has mechanical play, and pick-and-place nozzles add rotational offsets of their own. Fixturing (position correction) has to "straighten" the image before measuring, multi-camera stitching has to align several images into one coordinate system, and template matching often requires rescaling the image to a suitable resolution first — the common core of all these tasks is the **geometric transformation**: moving an image from one coordinate system to another. It looks like nothing more than "shift it a little, turn it a little," yet hidden inside are two questions that must be answered separately: **how do the coordinates map** — that is the matrix's job; **where do the new pixel values come from** — that is interpolation's job. Get either one wrong and the image develops holes, jagged edges, or a creeping blur — and blur is the sworn enemy of measurement accuracy.

The experiments in this chapter use the 480×360 synthetic scene shown in Figure 10.1. Every structure in it is deliberately designed to give geometric transforms a hard time: the text-like block in the upper left tests detail preservation, three 1 px bright horizontal lines and three 1 px dark vertical lines test thin-line survival, a 1 px diagonal line tests oblique structure, the central 80×80 checkerboard with 2 px squares is the high-frequency pattern most sensitive to resampling, and the ring in the upper right is for observing edge smoothness.

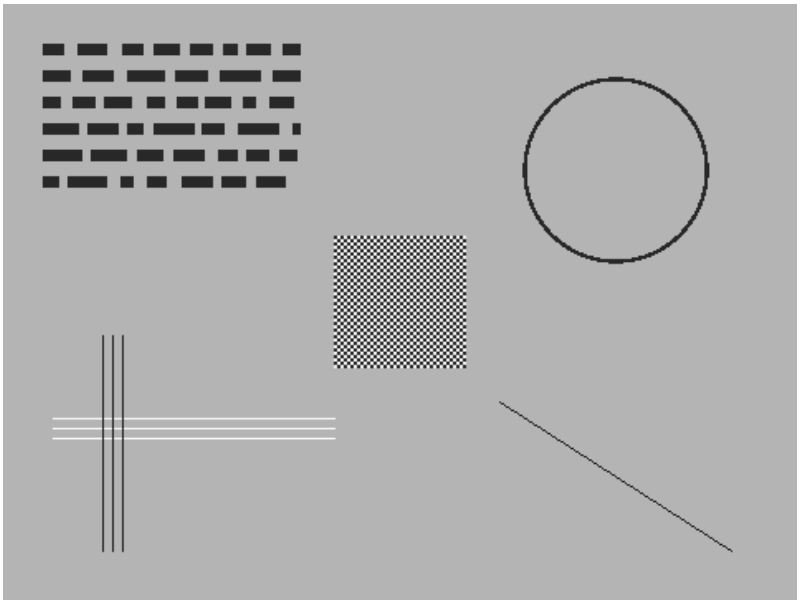


Figure 10.1: The test scene for this chapter (480×360): a text-like block, triplets of 1 px thin lines (horizontal/vertical), a 1 px diagonal line, a central checkerboard with 2 px squares, and a ring — all fine structures sensitive to interpolation and resampling.

10.1 Transformation Matrices

A two-dimensional point (x, y) is written in **homogeneous coordinates** as $(x, y, 1)^{\mathsf{T}}$ (see the linear algebra section of Chapter 2), whereupon translation, rotation, and scaling all unify into a single 3×3 matrix multiplication:

$$T(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}, \qquad R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Composing them arbitrarily (multiplying the matrices) yields the most general form, the **affine transformation**:

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

with 6 degrees of freedom in total; it keeps straight lines straight and parallel lines parallel. The entire action chain of fixturing — "first translate the workpiece to the center, then rotate it upright about the center" — is mathematically nothing but several such matrices multiplied into one.

With the matrix in hand, the next question is how to use it to generate the new image. The intuitive approach is **forward mapping**: walk over every pixel of the source image, compute where it lands in the destination image, and carry the gray value over. This approach has a fatal flaw — the landing coordinates are generally not integers, and after rounding, some destination pixels are fought over by multiple source pixels while others are never hit even once, leaving swaths of **holes**, especially conspicuous under rotation and magnification. The correct approach is exactly the reverse, called **inverse mapping**: walk over every pixel $\mathbf{p}'$ of the **destination** image and use the inverse matrix to find its source coordinates in the original image

$$\mathbf{p} = A^{-1}\,\mathbf{p}',$$

Here the real payoff of homogeneous coordinates shows itself: translation is inherently an addition, but after lifting one dimension it becomes a multiplication too, so **a transformation chain of any length can be pre-multiplied into a single matrix**. This seemingly trivial algebraic fact will turn into an engineering discipline that governs image quality in Section 10.3.

then go to the source image and "fetch" the gray value at this (generally non-integer) position. Every output pixel is assigned exactly once, and holes vanish at the root. The handwritten rotation in this chapter's companion code is implemented on exactly this principle. As for "how to fetch a gray value at a non-integer position" — that is precisely the topic of the next section.

10.2 Interpolation

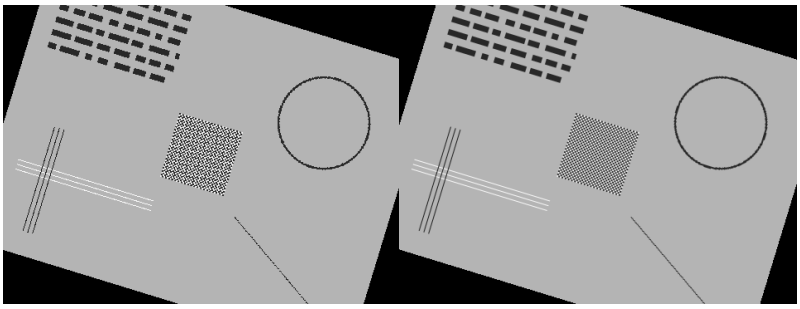
The source coordinates (s_x, s_y) computed by inverse mapping fall between the pixel grid points, so a gray value must be estimated from the neighboring pixels — this is **interpolation**. The two basic methods were already introduced in Chapter 2; here we only recap. **Nearest-neighbor interpolation** simply takes the nearest pixel after rounding; it produces no new gray values, at the cost of a position quantization error on the order of 0.5 px. **Bilinear interpolation** takes a distance-weighted average of the surrounding 4 pixels: writing $x_0 = \lfloor s_x \rfloor$, $y_0 = \lfloor s_y \rfloor$ and the fractional parts $a = s_x - x_0$, $b = s_y - y_0$, then

$$f(s_x, s_y) \approx (1-a)(1-b)\, f_{00} + a(1-b)\, f_{10} + (1-a)\, b\, f_{01} + ab\, f_{11},$$

where $f_{00}, f_{10}, f_{01}, f_{11}$ are the top-left, top-right, bottom-left, and bottom-right neighbors. It outputs continuously transitioning gray values — in essence a slight local smoothing.

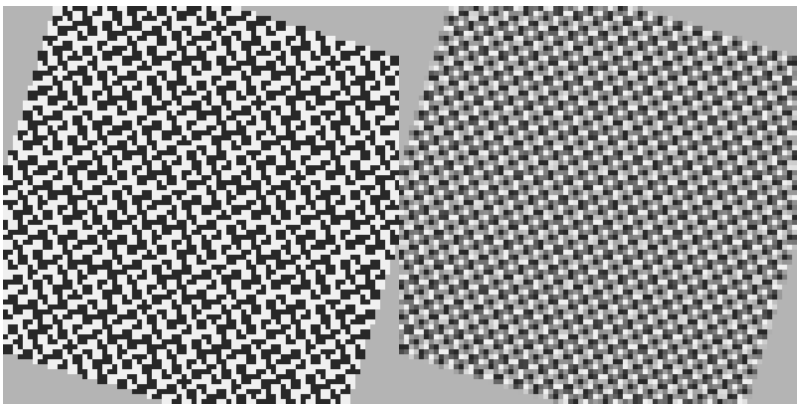
How much do they differ? We rotate the test scene by 17° with handwritten inverse-mapping implementations of nearest neighbor and bilinear (the two code paths are identical except for the sampling step); the results are in Figure 10.2. We then crop the central checkerboard region and magnify it 4 times for pixel-by-pixel viewing in Figure 10.3.

The difference is plain to see: the nearest-neighbor rotation is covered in jaggies, and because each output pixel "jumps" to its nearest source pixel, the black and white squares of the 2 px checkerboard are scrambled into irregular fragments; the bilinear result has smooth edges and tidy geometry, at the price



(a) Nearest neighbor (b) Bilinear

Figure 10.2: 17° rotation via handwritten inverse mapping: nearest neighbor vs. bilinear. At a glance the two look similar; the differences concentrate in the fine structures — see the magnified comparison in Figure 10.3.



(a) Nearest neighbor ×4 (b) Bilinear ×4

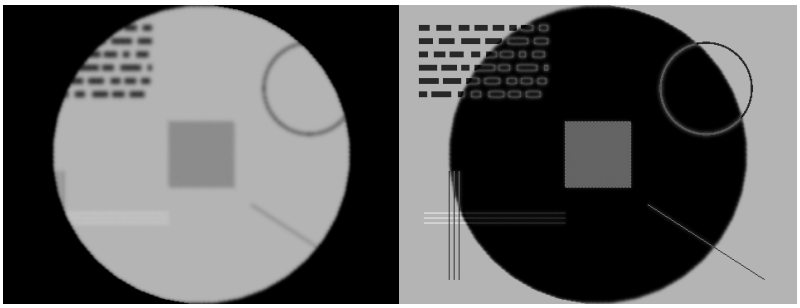
Figure 10.3: The central checkerboard region cropped and magnified 4 times. The nearest-neighbor result is visibly jagged, the black and white squares of the 2 px checkerboard scrambled by position quantization; the bilinear result has smooth edges, with the checkerboard transitioning into regular gray tones.

of introducing intermediate gray values that did not exist in the original, so fine structures go slightly soft.

10.3 The Cost of Resampling

The "slight smoothing" side effect of bilinear interpolation is almost invisible after a single pass, but every geometric transform **resamples** the image once, and the smoothing stacks up one layer at a time. To quantify this cost, we run an experiment that can "return to the starting point": rotate the test scene by 10° with bilinear interpolation, then by another 10°… 36 times in all, for a total angle of exactly 360° — the image should come back to its original place without a single discrepancy. The result is in Figure 10.4.

Rule for choosing the interpolation method: measurement tasks (edge localization, calipers, registration) use bilinear or higher-order interpolation, to keep nearest neighbor's position quantization from harming subpixel accuracy; **binary masks and label images must use nearest neighbor** — bilinear would interpolate a flood of intermediate gray values between 0 and 255, and the mask would no longer be a mask.



(a) 36 cumulative rotations of 10° each (b) Absolute difference from the original

Figure 10.4: The cumulative resampling experiment. (a) The image "back in place" after 36 bilinear rotations of 10°: the 2 px checkerboard has blurred into a uniform gray patch, the 1 px thin lines have all but vanished, the text block is reduced to a rounded shadow, and the four corners have been clipped to black by the rotations; (b) the difference image: discrepancies are clearly visible over the central structures, while the large bright regions in the corners come from frame clipping.

The position did indeed come back, but the image is unrecognizable: the 2 px checkerboard has blurred completely into a

uniform gray, the 1 px thin lines are reduced to faint traces, and every corner of the text block has been rounded off. Quantitatively, within the central disk of radius 170 px (avoiding the corner-clipped region), the mean absolute difference (MAD) between the 36-rotation image and the original is as high as **10.78** gray levels; as a control, a **single-step rotation by 360°** (mathematically the identity transform) has a MAD of **0.0000** — not a single byte inside the disk changed. Chaining the SDK's rotation interface 36 times gives 10.77, matching the handwritten implementation — proof that this is not a defect of any particular implementation but a property of resampling itself: **every resampling is an irreversible loss of information**.

From this follows the most important engineering discipline of the chapter: **compose transformation matrices freely in the math; never execute them step by step on the image**. When you need "translate first, then rotate, then scale," multiply the three matrices into one in code and perform **exactly one** inverse-mapping resampling — the result is mathematically equivalent to doing it in three steps, but in image quality the two are worlds apart.

The full-frame MAD is 78.57, far larger than the in-disk 10.78 — it is dominated by corner clipping: with `isChangeSize=false` the frame stays fixed, each rotation swings the corners out of frame and fills them with the fill color, and after 36 rotations only an inscribed disk survives at the center. When computing error statistics, be sure to exclude this region — otherwise you are measuring clipping, not interpolation.

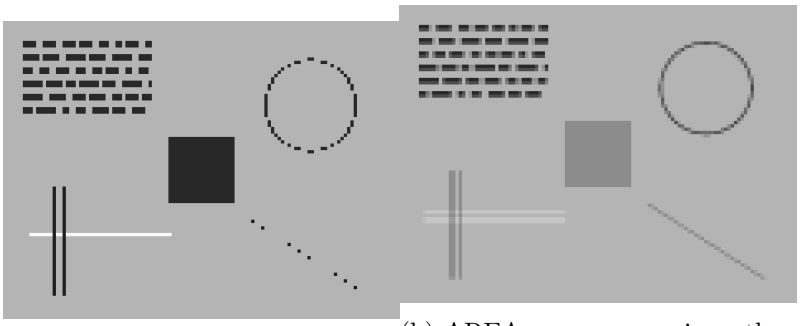
Corner clipping is itself a semantic issue to watch: the rotation interface's `isChangeSize` parameter decides whether the output frame is enlarged to contain the full bounding rectangle of the rotated image. With `false`, the frame matches the original, corner content is cut off, and the vacated regions are filled with the specified color; with `true`, the content is preserved losslessly but the frame grows and every downstream ROI coordinate is invalidated. Fixturing-style applications usually choose `false` and make sure the measured region lies inside the safe disk.

10.4 Scaling and Aliasing

Scaling is also a geometric transform, but **downsampling** has a trap that rotation does not. Shrinking an image by a factor of 4 means the sampling rate drops to 1/4; by the sampling theorem discussed in Chapter 1, structures in the original with a period shorter than 8 px simply cannot be represented at the

new sampling rate, and if left untreated they come back disguised as **aliasing** (see Chapter 11 for the frequency-domain explanation). The correct procedure is **prefilter, then decimate**: first filter out the components above the new Nyquist frequency, then reduce the resolution.

We compare using the SDK's `Sampling` interface: `NEAREST` mode amounts to taking 1 sample out of every 4 directly (no prefiltering), while `AREA` mode amounts to averaging each 4×4 source region (box prefiltering) before decimating. The results are in Figure 10.5 (for easier viewing, both downscaled images have been magnified back to the original size with nearest neighbor, which introduces no new information).



(a) NEAREST: direct decimation (b) AREA: area averaging, then decimation

Figure 10.5: Downsampling ×4 comparison. (a) Direct decimation: the 2 px checkerboard aliases into a solid dark patch, only one of the three 1 px bright lines randomly "survives," and the ring and the diagonal break into dashed lines; (b) box prefiltering followed by decimation: the checkerboard correctly averages to mid-gray, and all three thin lines are preserved — merely fainter, which is exactly what a thin line truly looks like once its energy has been spread out.

The direct-decimation result is shocking: the 2 px checkerboard aliases into a **solid dark patch** — the samples happen to land on the dark squares every time, and the high-frequency pattern masquerades as a low-frequency structure that never existed; of three identical 1 px bright lines only one "survives," the other

two happening to be skipped by the sampling, their fate determined entirely by their position relative to the sampling grid; the ring and the diagonal break into dashes. The prefiltered result keeps every structure: the checkerboard averages to mid-gray (its only honest representation at the lower resolution), and all three thin lines remain visible, merely fainter because their energy has been spread thin. **Downsampling must be prefiltered** — missed detections of thin line-like defects very often trace back to one "convenient" direct decimation in a thumbnail pipeline.

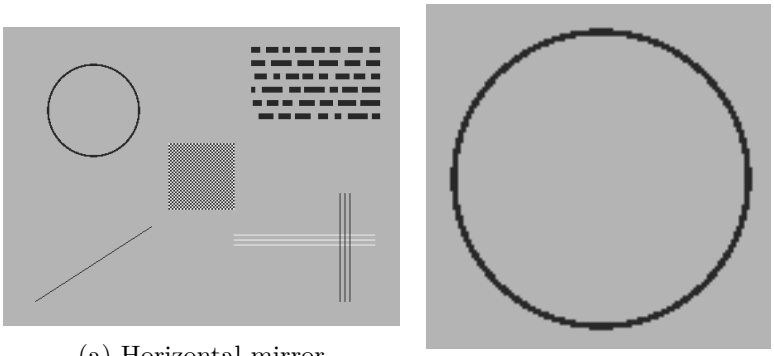
`Sampling` offers 6 interpolation modes; the applicable scenarios are tabulated below.

Mode	Meaning	Applicable scenarios
`LINEAR`	Bilinear	General-purpose default: magnification, mild shrinking
`NEAREST`	Nearest neighbor (direct decimation/copy)	Binary masks and label images; aliases when downsampling
`CUBIC`	Bicubic (4×4 neighborhood)	High-quality magnification, edges sharper than bilinear
`AREA`	Area averaging (built-in box prefiltering)	**First choice for downsampling**
`LANCZOS4`	8×8 Lanczos-windowed sinc	Highest-quality scaling; may ring near strong edges
`WEIGHT`	SDK custom weighted mode	Behaves close to LINEAR; verify by measurement before use

10.5 Mirroring, Cropping, and Stitching

A few common "integer-level" geometric operations remain; they involve no interpolation, move pixels byte by byte,

and lose no information. **Mirroring** performs horizontal (left-right) and vertical (top-bottom) flips, commonly used to unify the orientation of images from symmetric stations or images reflected by prisms. **Cropping** extracts an ROI region into a standalone small image, reducing the data volume for subsequent processing. Here is an engineering fact that must be remembered: the bottom-right corner of SciVision's `GenRect1` rectangular ROI is an **exclusive** endpoint — passing (305,35)–(434,164) yields a **129×129** output, not 130×130 (Figure 10.6). Writing code on the "both ends inclusive" intuition will systematically drop the last row and column. **Stitching** composes multi-camera or multi-view images into a large image; Section 5.12 of the SDK user manual provides the corresponding interfaces, which this book does not expand on.



(a) Horizontal mirror

(b) Cropping result (129×129)

Figure 10.6: Integer-level geometric operations. (a) Horizontal mirror: a left-right flip, moved byte by byte with no interpolation; (b) the ring region cropped with exclusive endpoints (305,35)–(434,164), actual output 129×129.

10.6 SciVision Implementation

Rotation and translation are provided by `SciSvRotationShift`, a single interface that does both jobs at once:

```
SCIMV::SciSvRotationShift rot;
SciColor fill(0, 0, 0);          // fill color for regions rotated out of frame
SciImage dst;
// rotate 17° about the image center, frame unchanged, translation (0,0)
long rc = rot.RotateShiftImage(src, fill, 17.0f,
                               /*isChangeSize=*/false, 0, 0, &dst);
```

The parameters are, in order: input image, fill color, rotation angle (degrees), `isChangeSize` (whether to enlarge the frame to contain the full rotated result; see Section 10.3 for the semantics), x/y translation amounts, and the output image. Two measured facts must be known. First, **this interface exposes no interpolation-method parameter**. We compared the SDK's rotation result for +17° against the handwritten implementation's **−17°** rotation (the SDK's positive-angle direction is opposite to the textbook convention — see the second fact; the negated angle is the same geometric transform; statistics taken within the central disk of radius 170): the MAD against handwritten bilinear at −17° is **1.05**, and against handwritten nearest neighbor at −17° is **3.90** — evidence that its interior is a bilinear-class smooth interpolation, safe to use in measurement pipelines; and for the same reason, this chapter's nearest-neighbor/bilinear comparison experiments were done with the handwritten inverse mapping (the SDK exposes no nearest-neighbor mode). Second, **the rotation direction for positive angles is opposite to the common counterclockwise convention**: the same +17° given to the SDK and to code implemented per the textbook convention rotates in opposite directions. When porting code from OpenCV or similar libraries, always verify the direction with an asymmetric image first, then settle the sign of the angle.

Scaling, mirroring, and cropping are invoked as follows:

```
SCIMV::SciSvSampling smp;
SciImage down;
smp.Sampling(src, src.Width() / 4, src.Height() / 4,
             SCI_SV_RESAMPLE_AREA, &down);   // use AREA for downsampling (built-in prefilteri

SCIMV::SciSvMirror mir;
```

```
SciImage flipped;
mir.Mirror(src, SCI_MIRROR_HOR, &flipped);   // horizontal mirror (left-right flip)

SCIMV::SciSvImageCropping cropper;
SciROI roi;
SciPoint tl(305, 35), br(434, 164);          // bottom-right corner is an exclusive endpoint
roi.GenRect1(tl, br);
SciImage patch;
cropper.ImageCrop(src, roi, 0, 0, &patch);   // output is 129×129
```

The parameters of `Sampling` are the input image, target width, target height, interpolation mode (see the table in Section 10.4 for choosing among the 6 modes), and the output image; the second parameter of `Mirror` is the mirror type; the third and fourth parameters of `ImageCrop` are the cropping type and the out-of-bounds fill gray value. All return codes should be checked. The complete project that generates all of this chapter's experimental images is located at `code/geometric_transforms/`.

Industry Case: The Invisible Blur in a Fixturing Chain

A placement-inspection project ran the pipeline "template localization → rotation correction → caliper measurement." During debugging, to make the translation amount and the angle easier to verify separately, an engineer split the correction into two steps: translate once, then rotate once — two resamplings. After the line went into production, the caliper edges' transition band blurred from about 2 px to 3 px and edge-localization repeatability degraded by roughly thirty percent; checks of the lighting, the lens, and vibration all came up empty. The cause was finally found in the image chain: two bilinear resamplings had stacked two layers of smoothing. After multiplying the translation and rotation matrices into one affine matrix in code and resampling only once, the edge width and repeatability recovered. The deeper lesson goes further: **skip the correction entirely when you can** — if the downstream tools support pose-aware ROIs, let the ROI rotate with the workpiece (see Chapter 19) and the image never needs to be resampled at all.

10.7 Summary

- **Geometric transform = matrix + interpolation**: homogeneous coordinates unify translation, rotation, scaling, and affine transforms into 3×3 matrices; generating the image requires inverse mapping — each output pixel samples back into the source image, whereas forward mapping leaves holes.
- **Choose the interpolation method by task**: measurement tasks use bilinear or higher order; binary masks and label images must use nearest neighbor (it introduces no new gray values).
- **Every resampling is a loss of information**: after 36 rotations of 10° back to the starting position the MAD reaches 10.78, while the single-step equivalent transform gives 0.0000. Multiply a transformation chain into one matrix first and resample only once; if a pose-aware ROI will do, don't touch the image.
- **Downsampling must be prefiltered**: direct decimation aliases high-frequency structures (the checkerboard turns into a solid patch, thin lines vanish at random); use modes with built-in prefiltering such as AREA.
- **Verify the engineering semantics**: `RotateShiftImage` exposes no interpolation parameter (measured to be bilinear-class) and its positive-angle direction is opposite to the common convention; the bottom-right corner of `GenRect1` is an exclusive endpoint — (305,35)–(434,164) crops a 129×129 region.

The standard treatment of geometric transforms and resampling interpolation is the textbook by Gonzalez and Woods (Gonzalez and Woods 2018); the widely used cubic convolution interpolation, whose accuracy sits between bilinear and spline interpolation, originates in the paper by Keys (Keys 1981). For a more systematic treatment of geometric transforms and interpolation in measurement systems (including the relationship between projective transforms and subpixel accuracy), see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

11 Frequency Domain Processing and the FFT

In the previous chapters we have worked entirely in the spatial domain: an image is an array of pixels, filtering is a weighted operation over a neighborhood, and every question revolves around "what is the gray value **where**." This chapter switches perspective — the frequency domain cares not about "where" but about "**how fast**": how rapidly does the image's gray value vary across space? A slowly undulating background is low frequency; sharp edges and fine, dense textures are high frequency. This perspective is no mathematical game — it solves practical problems that leave the spatial domain helpless. Imagine a production-line image contaminated by periodic electromagnetic interference: the entire image is overlaid with fine diagonal stripes. In the spatial domain this interference is everywhere, entangled with the image content; mean filtering cannot wipe it away, and median filtering is equally powerless. But in the frequency domain, this whole-image interference **collapses into two isolated bright dots** — carve them out with a small circle, and the interference vanishes almost losslessly. This chapter will demonstrate this "killer" application in full; but first, let us learn the language of the frequency domain.

Our experimental scene (Figure 11.1) is a 512×512 synthetic image: a large dark rectangle and a large circle provide low-frequency blocky structure, the middle band is a vertical stripe pattern with a period of exactly 8 px (a high-frequency component of known frequency), and below it sit the dot-matrix letters "FFT" (text-like detail). Each component will leave a recognizable signature in the spectrum.

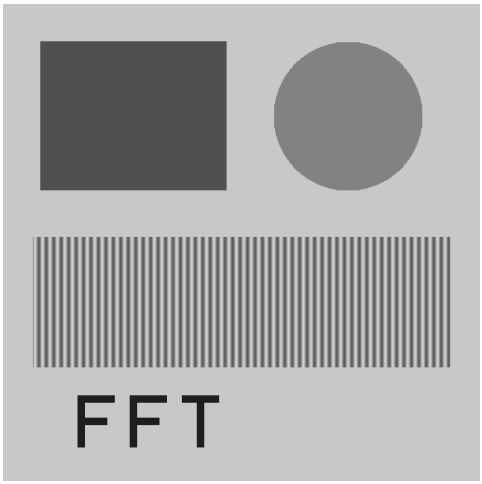


Figure 11.1: The 512×512 experimental scene: a large rectangle and a large circle (low-frequency blocky structure), a vertical stripe band with an 8 px period (known high frequency), and the dot-matrix text "FFT" (detail component).

11.1 The Two-Dimensional Fourier Transform

The core idea of the Fourier transform is that any signal can be decomposed into a superposition of sinusoids of different frequencies. For a $W \times H$ discrete image $f[n, m]$ (with n the row and m the column), the **2D Discrete Fourier Transform (DFT)** is defined as

$$F[u, v] = \sum_{n=0}^{H-1} \sum_{m=0}^{W-1} f[n, m]\, e^{-j2\pi\left(\frac{um}{W} + \frac{vn}{H}\right)},$$

where u, v are the frequency indices in the horizontal and vertical directions. The inverse DFT superposes the frequency components back into the image:

$$f[n, m] = \frac{1}{WH} \sum_{u=0}^{W-1} \sum_{v=0}^{H-1} F[u, v]\, e^{+j2\pi\left(\frac{um}{W} + \frac{vn}{H}\right)}.$$

$F[u, v]$ is a complex number, and in engineering practice it is almost always split into two more intuitive quantities. The **magnitude spectrum** $|F[u, v]| = \sqrt{\mathrm{Re}^2 + \mathrm{Im}^2}$ answers "**how much** of the sinusoidal component at frequency (u, v) is present"; the **phase spectrum** $\arg F[u, v]$ answers "**where** that component sits" — the same set of sinusoids, given different phases, superposes into a completely different image. Filtering operations work mainly on the magnitude, but when reconstructing the image the phase must be preserved exactly as it is, or the image structure will become unrecognizable.

One frequency component deserves to be singled out: at $u = v = 0$ the exponential term is identically 1, so $F[0, 0] = \sum f[n, m]$ — exactly the sum of all gray values in the image, i.e. WH times the **average gray level**. This is the **DC component** — the statement in Chapter 6 that "the sum of the kernel weights equals the DC gain" originates exactly here: a filter with DC gain 1 leaves $F[0, 0]$ unchanged, and therefore leaves the image's average brightness unchanged.

Plotting $|F|$ directly as an image shows almost nothing: the energy of the DC component is orders of magnitude larger than that of nearly all high-frequency components, so under a linear gray mapping the entire spectrum is essentially black except for a single point at the center. The standard practice is to display the **log spectrum** $\log(1 + |F|)$, which compresses the enormous dynamic range into an interval the eye can resolve — every spectrum in this chapter is plotted this way.

Computing the DFT directly from the definition takes $O(N^2)$ operations ($N = WH$) — about 7×10^{10} for a 512×512 image, which is unacceptable. The **Fast Fourier Transform (FFT)** exploits the symmetry of the twiddle factors to bring the complexity down to $O(N \log N)$. This is precisely what makes frequency-domain methods practical, and it is why this chapter's experiments use image sizes that are integer powers of 2.

11.2 Reading the Spectrum

Figure 11.2 is the log magnitude spectrum of the experimental scene (with the DC component shifted to the image center). Reading a spectrum is a skill you can pick up quickly — let us identify each feature against the scene.

- **The bright central blob**: low-frequency energy. Slowly varying blocky structures like the large rectangle and the large circle concentrate nearly all of their energy near the center.

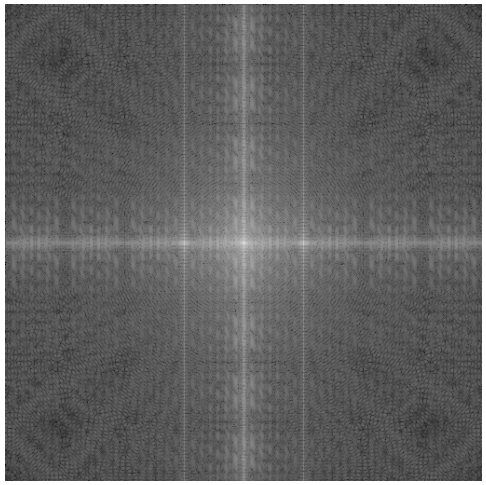


Figure 11.2: Log magnitude spectrum of the experimental scene (DC centered). The bright central blob is low-frequency energy, the bright horizontal/vertical cross comes from the axis-aligned edges of the rectangle and the stripe band, and the symmetric pair of dots at ± 64 on the horizontal axis is the signature of the 8 px vertical stripes.

- **The bright horizontal and vertical cross**: the rectangle's horizontal/vertical edges and the stripe band's horizontal boundaries are all "abrupt transitions along one direction," and in the frequency domain an abrupt transition spreads out into a line **perpendicular to the edge**.
- **A symmetric pair of bright dots on the horizontal axis**: a pure periodic stripe pattern is, in the frequency domain, just a pair of impulses — this is the most important correspondence in this chapter.

The position of this pair of dots can be **predicted first, then verified**. A stripe pattern with period T pixels repeats exactly W/T times across an image of width W, so its spectral peak appears at horizontal frequency index

$$u = \frac{W}{T} = \frac{512}{8} = 64$$

— that is, at ± 64 on either side of the center. In the experiment we exclude the central low-frequency region and search the whole spectrum for the maximum magnitude; the measured peak position is $(\Delta u, \Delta v) = (-64, 0)$ — **an exact hit on the prediction**. This "known period $\rightarrow$ spectral peak position" conversion is a directly usable weapon when troubleshooting periodic interference on a production line: measure the pixel period of the interference stripes, and you know exactly where in the spectrum to look for them.

11.3 Frequency-Domain Filtering

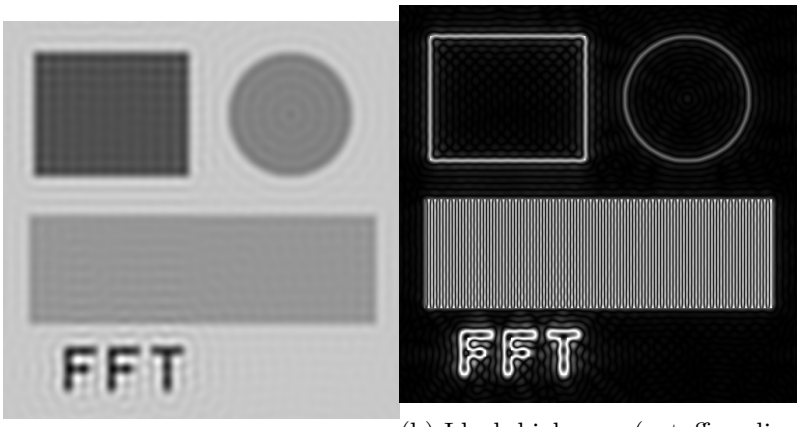
The theoretical cornerstone of frequency-domain filtering is the **convolution theorem**: convolution in the spatial domain is equivalent to pointwise multiplication in the frequency domain,

$$f * h \;\longleftrightarrow\; F \cdot H.$$

Conjugate symmetry: the spectrum of a real-valued image satisfies $F[-u, -v] = F^*[u, v]$, so the magnitude spectrum is symmetric about the center — this is why spectral peaks always appear in **pairs**. When we do notch filtering later, the conjugate peak of every interference peak must be handled along with it, or the result of the inverse transform will no longer be a real image.

This means that all of the linear filtering of Chapter 6 can be carried out in the frequency domain: take the FFT of the image, multiply the spectrum pointwise by the filter's frequency response H, then apply the inverse transform. The two routes are **mathematically identical**; the engineering trade-off is computational cost. Spatial convolution with a $K \times K$ kernel costs $O(K^2)$ multiply-adds per pixel, while the frequency-domain route's cost is independent of kernel size — use the spatial domain for small kernels, and the frequency domain for large ones (tens of pixels and up).

The simplest frequency-domain filter is the **ideal lowpass filter**: $H = 1$ within radius D_0 of the DC center, $H = 0$ outside, zeroing out the high frequencies in one clean cut; the **ideal highpass** is exactly the opposite. We run the experiment with a cutoff radius of 30 px (normalized frequency $30/256 \approx 0.1172$); the results are shown in Figure 11.3.



(a) Ideal lowpass (cutoff radius 30)

(b) Ideal highpass (cutoff radius 30)

Figure 11.3: Results of ideal lowpass and highpass filtering. (a) Lowpass: the 8 px stripes (frequency $64 > 30$) are flattened wholesale into a uniform gray block, the "FFT" text is blurred, and **ring after ring of wave-like ringing appears around every edge**; (b) highpass: the interiors of the blocky structures are emptied out, leaving only edge outlines, while the stripe band — its frequency above the cutoff — is preserved intact.

The lowpass result confirms the prediction: the stripe frequency of 64 lies far outside the cutoff radius of 30, and the entire stripe band is flattened into uniform gray. But notice the concentric ripples around the rectangle, the circle, and the text — this is **Gibbs ringing**, the price the ideal filter must pay. The cause is clear-cut: a hard rectangular truncation in the frequency domain has the sinc function as its spatial-domain counterpart, and the sinc has oscillating sidelobes that trail on endlessly; by the convolution theorem, multiplying in the frequency domain equals convolving in the spatial domain with this long-tailed kernel, so every edge gets dragged out into a train of ripples. **For this reason the ideal filter is rarely used in engineering practice**; one uses Gaussian or Butterworth filters instead — their frequency responses transition smoothly, their spatial-domain kernels no longer oscillate, and the only cost is a cutoff edge that is not quite as "sharp."

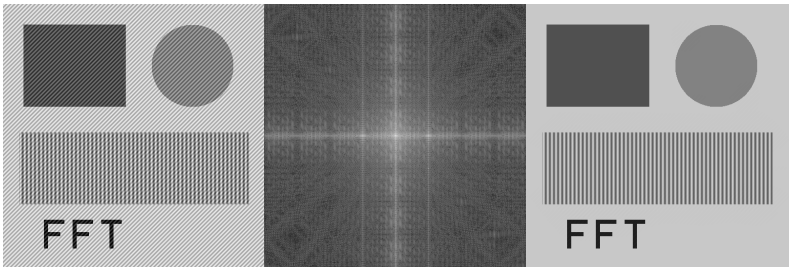
The highpass result likewise speaks the language of the spectrum: once the low frequencies within radius 30 are zeroed, the blocky structures are reduced to their edge outlines (edges are high frequency), while the stripe band survives almost untouched — its frequency of 64 is above the cutoff of 30, inside the highpass passband. Highpass filtering is therefore often used as a means of edge enhancement and background suppression.

The lesson of ringing fits in one sentence: **the harder you cut in the frequency domain, the longer the smear in the spatial domain**. This is the flip side of the same coin as Chapter 6's observation that "the Gaussian kernel's frequency response has no sidelobes, so its smoothing is clean."

11.4 Notch Filtering of Periodic Noise

Now for this chapter's central demonstration. We superimpose a diagonal sinusoidal interference $40\sin(2\pi(r+c)/8)$ onto the scene (period about 5.7 px along the diagonal), simulating periodic stripe contamination from electromagnetic interference or mechanical vibration. The RMSE between the noisy and clean images is **28.13** — consistent with the theoretical value $40/\sqrt{2} \approx 28.3$ (the RMS amplitude of a sinusoid). Spatial-domain filtering is powerless against it: the interference's spatial frequency is quite close to that of the useful 8 px stripes in the scene, and any smoothing strong enough to suppress the interference would wipe out the useful stripes along with it.

But in the frequency domain, this interference is just **two dots**. Figure 11.4 shows the complete pipeline.



(a) With periodic noise (b) Spectrum of the noisy image (c) Notch-filtered result

Figure 11.4: Notch removal of periodic noise. (a) The whole image is covered by fine diagonal stripes, RMSE 28.13; (b) in the spectrum the interference collapses into a conjugate pair of bright dots on the diagonal at $(\pm 64, \pm 64)$; (c) after carving out the two dots with small circles of radius 5 and inverse-transforming, the stripe interference is gone, the RMSE drops to 0.61, and the useful 8 px vertical stripes are unscathed.

Predict first: the interference has a period of 8 px along both the row and column directions, so the spectral peaks should appear at $(\pm 512/8, \pm 512/8) = (\pm 64, \pm 64)$. Searching the spectrum yields $(-64, -64)$ and its conjugate peak $(+64, +64)$ — another exact hit. Next comes the four-step **notch filtering** procedure:

1. Take the FFT of the noisy image (DC centered);
2. Exclude the central low-frequency region and search for the interference peak of maximum magnitude;
3. **Zero out** the complex spectrum inside a small circle of radius 5 px centered on the peak, and treat the conjugate peak the same way;
4. Inverse-transform back to the spatial domain and clamp to [0, 255].

The result (Figure 11.4c) is nearly indistinguishable from the original: the RMSE drops from 28.13 to **0.61**, more than 98%

of the interference energy is eliminated, and the useful vertical stripes (peaks at ± 64 on the horizontal axis, far away from the interference peaks) are left without a scratch. This is exactly what makes frequency-domain methods irreplaceable: **two components can be very close in spatial-domain "speed," yet as long as their direction or period differs slightly, they are separate points in the spectrum** — spatial filtering cuts by "speed" alone, while a frequency-domain notch strikes precisely where you aim. The residual 0.61 comes from the small amount of the image's own energy lost inside the carved-out circles — a reminder that for the notch radius, smaller is safer.

11.5 The Sampling Theorem Revisited

Armed with the language of the frequency domain, the aliasing intuition from Chapter 1 — "sample each period at least twice" — can finally be upgraded to a formal statement. The **Nyquist sampling theorem**: as long as the signal's highest frequency component is below half the sampling frequency, the original signal can be perfectly reconstructed from its samples. The frequency domain makes it clearest — sampling replicates the original signal's spectrum periodically at intervals of the sampling frequency, and once the signal's bandwidth exceeds half the sampling frequency, adjacent spectral replicas **shift into overlap**: high-frequency components masquerade as low frequencies and mix into the signal, with no way to separate them after the fact. This also explains the rule from Chapter 10 that an image must be smoothed before shrinking: downsampling amounts to lowering the sampling frequency, and **pre-filtering means trimming the spectrum down to within the new Nyquist limit first** — better to lose detail than to let detail turn into spurious low-frequency artifacts.

11.6 SciVision Implementation

All of this chapter's experiments are carried out by the `SCIMV::SciSvFFT` class; the core calls are as follows:

```
SCIMV::SciSvFFT fft;
SciImage fftImg;
// mode=0: shift the DC component to the spectrum center; fftImg is a 2-channel 32F image with
fft.ApplyFFTGeneric(src, 0, &fftImg, NULL);

// Ideal lowpass/highpass: frequency is the normalized cutoff (relative to the half-width W/2)
float cutoff = 30.0f / (512 / 2);
SciImage lp;
fft.GenerateLowpass(cutoff, 0, 512, 512, &lp);

// The convolution theorem in action: multiply the spectrum pointwise by the filter
SciImage fftLP, out32;
fft.ApplyFFTConvolution(fftImg, lp, &fftLP);
// Inverse transform: always request SCI_IMAGE_32F, then clamp to [0,255] yourself
fft.ApplyIFFTGeneric(fftLP, 0, &out32, true, SCI_IMAGE_32F);
```

The complex spectrum output by `ApplyFFTGeneric` can be read and written directly through `ImageData()` and `Step()` — the real and imaginary parts at row r, column c sit at float offsets `r*stride + 2*c` and `r*stride + 2*c + 1`. The notch filtering of Section 11.4 is implemented exactly this way, manually zeroing the complex values inside the two small circles — the SDK provides no ready-made notch interface, and none is needed.

There are three experimentally confirmed pitfalls in using this API, recorded as found:

- **IFFT output as 8U is min-max normalized**: requesting 8-bit output directly stretches the gray values wholesale, and the roundtrip (FFT→IFFT) RMSE comes out at a staggering 33.62 — it looks like the algorithm is broken, but it is the normalization. Request `SCI_IMAGE_32F` to get the raw floating-point result and clamp it yourself, and the roundtrip RMSE is **0.0000** — the transform itself is lossless.
- **The magnitude image built into `ApplyFFTGeneric` is nearly all black**: its `imageMagnitude` output uses linear normalization, the DC component dominates, and all other frequencies are invisible. The log spectrum $\log(1 + |F|)$ must be computed yourself from the complex data.

- **`RemovePeriodicPatternsByFFT` is unusable**: this interface is explicitly marked "not implemented" in the header file; for periodic noise removal, do the notch filtering manually following this chapter's procedure.

The complete project that generates all of this chapter's images is located at `code/fourier/`; the predicted versus measured spectral peak positions are printed directly to the console.

Industry Case: Weave Patterns on Textile Surfaces

In a textile surface-defect inspection project, the fabric's own warp-and-weft weave pattern was a strong periodic structure whose gray-level fluctuation far exceeded the defect signals — broken yarns, oil stains — leaving the defects nearly invisible in the raw image. Spatial filtering was caught in a dilemma: a small kernel could not filter the weave out cleanly, while a large kernel blurred the defects away together with the weave. Switching to a frequency-domain approach dissolved the problem — the weave's fundamental frequency and its harmonics form a set of bright dot pairs at stable positions in the spectrum; applying a small-radius notch to each pair and inverse-transforming back to the spatial domain removed the weave wholesale, leaving the defects plainly visible in the residual image. The key lesson from on-site tuning was that **for the notch radius, smaller is safer**: defect energy is also distributed near the weave peaks, and while a larger radius filters the weave more thoroughly, it swallows defect energy and lowers detection contrast — size the notch to just cover the spectral peak.

11.7 Summary

- **The frequency domain answers "how fast"; the spatial domain answers "where"**: the magnitude spectrum gives how much of each frequency component is present, the phase spectrum gives where they sit; the DC component $F[0, 0]$ is the sum of all gray values, and the log display $\log(1 + |F|)$ is the standard way to look at a spectrum.

- **Spectra can be read — and, better still, predicted**: stripes with period T in an image of width W produce a conjugate peak pair at $u = W/T$ — the 8 px stripes predicted ± 64, and the measurement hit it exactly.
- **The convolution theorem $f * h \leftrightarrow F \cdot H$ connects the two domains**: use spatial convolution for small kernels, the frequency-domain route for large ones; a hard cut in the frequency domain invites Gibbs ringing (the sinc's trailing tail), so engineering practice replaces ideal filters with smoothly transitioning Gaussian/Butterworth ones.
- **Notch filtering is the specific cure for periodic interference**: whole-image periodic contamination is just a few points in the spectrum; zero out small circles and inverse-transform — in the experiment the RMSE dropped from 28.13 to 0.61, something no spatial filter can achieve.
- **The Nyquist theorem, stated in frequency-domain terms**: the signal's highest frequency must be below half the sampling frequency, or shifted spectral replicas overlap and produce aliasing — this is the root reason prefiltering is mandatory before downsampling.

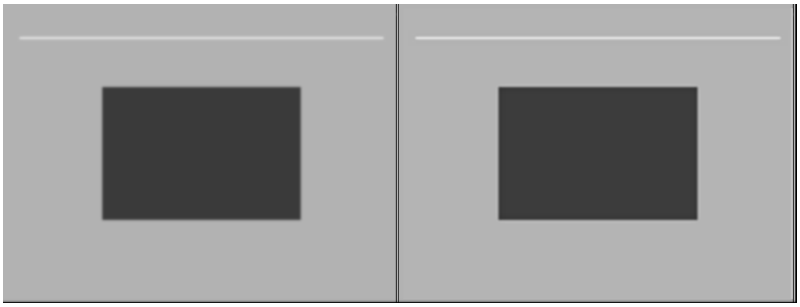
The fast algorithm that made the FFT practical originates in the classic 1965 paper by Cooley and Tukey (Cooley and Tukey 1965), which cut the complexity of the DFT from $O(N^2)$ to $O(N \log N)$; the systematic treatment of the Fourier transform in digital image processing (including frequency-domain filtering and notch filters) is the textbook by Gonzalez and Woods (Gonzalez and Woods 2018). For a systematic treatment of frequency-domain methods in industrial image processing (including optimal filtering and frequency-domain features), see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

12 Advanced Enhancement

Up to this point, most of what we have done to images is "repair": removing noise (Chapter 6), flattening illumination (Chapter 4), stretching gray values into a suitable range. What these operations share is restoration — whatever the image was supposed to look like, we patch it back to that state. This chapter turns in a different direction: **reinforcement**. Some information truly is in the image, yet too weak for downstream algorithms to use directly — edges flattened by the softening of the lens and defocus, dark-region detail drowned in readout noise, bright-region detail crushed into a sheet of white by the sensor's full-well and quantization ceiling. The two tools of this chapter exist precisely for this: **sharpening** stands the flattened edges back up, compensating for the MTF losses of the imaging chain; **high dynamic range (HDR) imaging** uses multiple exposures to recover both the bright and dark ends of information that exceed the dynamic range of a single frame. Figure 12.1 gives a first look at the intuitive effect of the first tool: an edge softened by defocus is stood back up by unsharp-mask enhancement. As for shadows caused by uneven illumination, Chapter 4 has already discussed shading correction in detail, and we will not repeat it here.

12.1 Sharpening and the Unsharp Mask

The most classic — and most widely used — form of sharpening is **unsharp masking**. The seemingly self-contradictory name comes from the darkroom era: photographers deliberately printed a blurred ("unsharp") copy of the negative as a mask and exposed it together with the original, and the result was, counterintuitively, a sharper photograph. The digital version takes only three steps: blur the image once with a low-pass



(a) Softened scene (defocus simulated by a $\sigma = 1.5$ Gaussian)

(b) Unsharp-mask enhancement ($\lambda = 1.0$)

Figure 12.1: Unsharp-mask sharpening. (a) A sharp scene softened by a $\sigma = 1.5$ Gaussian, with the step edges dragged out into transition bands; (b) moderate enhancement at $\lambda = 1.0$ stands the edges back up without introducing visible artifacts.

filter, subtract the blurred version from the original to obtain a detail layer, then add the detail layer back in proportion:

$$g = f + \lambda\,(f - G_\sigma * f),$$

where $G_\sigma * f$ is the Gaussian blur of f (in Chapter 6 we used it to suppress noise; here we use it to separate frequency components) and λ is the enhancement strength. The logic of the operation is perfectly direct: the Gaussian blur keeps the low frequencies and filters out the high ones, so $f - G_\sigma * f$ is exactly the high-frequency part that was filtered away — edges, thin lines, texture; multiplying it by λ and adding it back to the original is equivalent to amplifying the image's high-frequency content by a factor of $1+\lambda$. λ is the "high-frequency gain knob": $\lambda = 0$ passes the image through unchanged, and the larger λ, the stronger the edge contrast. The σ of the blur kernel decides "how fine counts as detail" — it should be comparable to the blur scale being compensated.

We demonstrate with a controlled experiment. Take the same synthetic scene as in Chapter 6 (bright background 180, dark rectangle 60, one thin bright line 2 px wide at 250), and first

Sharpening compensates for the MTF attenuation caused by lens aberrations, slight defocus, and the like (Chapter 3), but it **cannot create information**: it can only amplify whatever high-frequency residue still survives, together with the noise inside it. If the MTF at some spatial frequency has already decayed to zero, the detail layer contains no trace of it whatsoever, and no value of λ will conjure it back. Sharpening is an "amplifier", not a "restorer".

soften it with SciVision's Gaussian filter (9×9, $\sigma = 1.5$) to simulate a slightly defocused lens: this yields Figure 12.1a, where the edges are already visibly fuzzy. Then, using the same Gaussian (9×9, $\sigma = 1.5$) as the blur term of the unsharp mask, we enhance with $\lambda = 1.0$ and obtain Figure 12.1b: the step edges become crisp again, and the contrast of the thin bright line recovers substantially.

12.2 Overshoot and Ringing

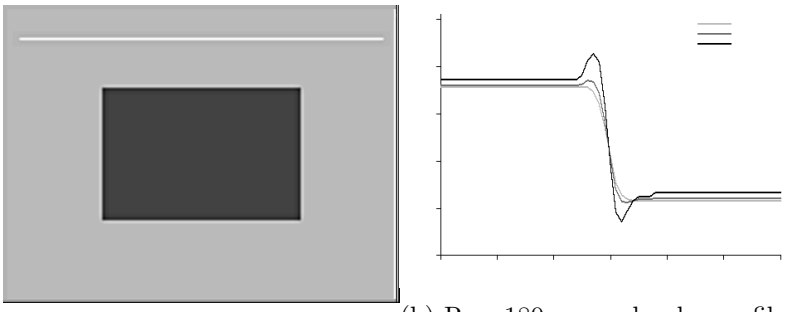
Since λ is a gain knob, would turning it up make things even sharper? Figure 12.2a shows the result at $\lambda = 4.0$: the edges are indeed "sharper", but a glaring bright halo floats around the dark rectangle, and the inside of the rectangle is rimmed with a band that is too dark. These are **overshoot** and **undershoot** — the sharpening "pushes too hard" upward and downward on the two sides of the step, driving the gray values past their original plateau levels. When the enhancement kernel has a steep frequency cutoff, the overshoot can also repeat as a decaying oscillation, called **ringing**.

Numbers are the most persuasive. We extract the gray-level profile along row 180, columns 90–150 (cutting straight across the left edge of the dark rectangle, with nominal gray values of 180 on the bright side and 60 on the dark side); the three curves are plotted in Figure 12.2b, with statistics as follows:

Table 12.1: Overshoot statistics of the row-180 profile (columns 90–150); nominal gray values 180/60

Profile	min	max	Overshoot / undershoot
Softened original	58	178	+0 / −2
$\lambda = 1.0$	56	185	+5 / −4
$\lambda = 4.0$	36	213	**+33 / −24**

At $\lambda = 1.0$ the overshoot is only $+5/-4$ gray levels — the same order as the image noise and invisible to the eye; at $\lambda = 4.0$ it surges to $+33/-24$, pushing out "false plateaus" of roughly 30 gray levels on each side of the edge.



(a) Over-enhancement ($\lambda = 4.0$) (b) Row-180 gray-level profiles compared

Figure 12.2: The overshoot experiment. (a) $\lambda = 4.0$ produces a pronounced bright halo outside the dark rectangle; (b) profiles over columns 90–150 (light gray = softened original, mid gray = $\lambda = 1.0$, black = $\lambda = 4.0$), with the edge near column 120: the $\lambda = 4.0$ curve shoots to 213 and 36 on the two sides of the step, far beyond the nominal plateaus of 180/60.

Overshoot is by no means merely an aesthetic issue; it does real damage to both measurement and inspection. **For measurement**: the caliper tools of Chapter 20 locate edges at the derivative extrema of the gray-level profile; overshoot changes the shape of the profile on both sides of the edge, the derivative curve is distorted accordingly, and the edge positions found shift systematically — you thought sharpening made the edge "easier to find", when in fact it "moved the edge". **For inspection**: to blob analysis or threshold segmentation, the bright halo around a dark workpiece is a ring of genuinely existing high-brightness area, enough to trigger a false defect alarm; conversely, the dark rim inside a bright region may be misjudged as a crack. Hence the rule for choosing λ: **extract edge profiles on representative samples, increase λ until the overshoot amplitude approaches the image noise amplitude, and give it no more**. Once the overshoot is buried in the noise, it has no systematic effect on downstream algorithms; in this example $\lambda = 1.0$ (overshoot +5, the same level as the noise) passes, while $\lambda = 4.0$ is far over the line. Also watch out for saturation: once overshoot is clipped at 0 or 255,

the original gray-level information is lost for good.

12.3 High Dynamic Range and Exposure Fusion

The second kind of "weak information" problem lies in **dynamic range**: the ratio of irradiance between the brightest and darkest parts of the scene exceeds what a single frame can record. The ceiling comes from the sensor's full-well capacity and 8-bit quantization (Chapter 3, Chapter 1); the floor comes from readout noise. The result is a dilemma: expose for the highlights and the shadow detail sinks into noise; expose for the shadows and the highlights wash out to solid white. The workpiece contour next to a welding pool, or characters etched on polished metal, are typical victims.

The idea of **multi-exposure fusion (exposure fusion)** is: if the dynamic range is not enough, shoot several frames and combine them. Capture the same static scene at multiple exposure levels — short exposures keep the highlights, long exposures scoop up the shadows — then merge the information pixel by pixel into a single estimated **irradiance** map. Under the linear-response assumption (pixel value $v_i \approx E \cdot t_i$, where t_i is the relative exposure), every frame gives one observation v_i/t_i of the irradiance, and a weighted average yields the estimate:

$$\hat{E}(p) = \frac{\sum_i w(v_i)\, v_i/t_i}{\sum_i w(v_i)}, \qquad w(v) = \min(v,\, 255 - v) + \varepsilon.$$

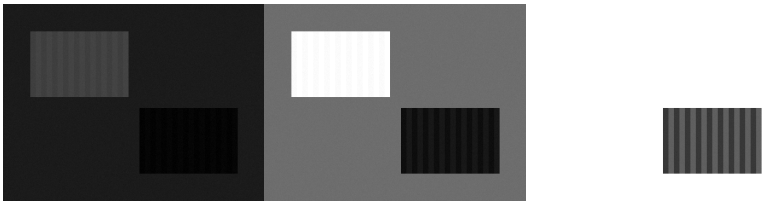
This is a simplified form of Debevec-style irradiance recovery, and w is the **triangle weight**: the weight is largest when the pixel value sits in the middle and falls toward zero as the value approaches 0 or 255.

The recovered $\hat{E}$ exceeds the 8-bit range, so a final **tone mapping** step compresses it back into the displayable interval. This chapter uses the simplest global γ mapping: $\text{out} = 255\,(\hat{E}/\hat{E}_{\max})^{1/2.2}$, where $\gamma = 1/2.2$ lifts the shadows

The intuition behind the triangle weight is an election with vote screening: each exposure frame casts one vote on the pixel's irradiance, but **votes from too-dark pixels don't count** (the signal is drowned in noise, so v_i/t_i is wildly unreliable), and **votes from too-bright pixels don't count either** (already clipped to saturation, so v_i simply does not equal $E \cdot t_i$). Only frames with a moderate exposure get a say — and for any point in the scene, some exposure level is bound to be just right, which is the entire point of taking multiple exposures.

and compresses the highlights, roughly matching human visual perception.

The irradiance field of the experimental scene is constructed as follows: a mid-gray background of 110; in the upper-left bright region, hidden vertical stripes of 252/272 (contrast 20, but with the peak above the 8-bit ceiling of 255); in the lower-right dark region, hidden vertical stripes of 14/24 (contrast 10, hugging the noise floor). We expose at three gain levels, $\times 0.25$ / $\times 1.0$ / $\times 4.0$, each clipped to [0,255] with $\sigma = 2$ readout noise added (fixed seeds 11/22/33 for reproducibility), giving the three frames of Figure 12.3; fusing and tone-mapping yields Figure 12.4.



(a) Short exposure ($\times 0.25$) (b) Mid exposure ($\times 1.0$) (c) Long exposure ($\times 4.0$)

Figure 12.3: Three exposure levels. (a) $\times 0.25$: the bright-region stripes survive while the dark region sinks into noise; (b) $\times 1.0$: the bright-region stripes are almost entirely clipped at 255, and the dark-region detail is barely discernible; (c) $\times 4.0$: the dark-region stripes are clear, but the bright region is completely saturated into solid white.

Beyond eyeballing, we compute the mean and standard deviation over the two stripe-detail patches (the standard deviation measures the visibility of the stripe contrast):

Table 12.2: Mean and standard deviation of the detail patches compared

Image	Dark patch mean / std	Bright patch mean / std
Short exposure ($\times 0.25$)	4.8 / 2.35	65.5 / 3.22
Mid exposure ($\times 1.0$)	19.0 / 5.40	253.5 / 2.03
Long exposure ($\times 4.0$)	76.0 / 20.13	255.0 / 0.00

Image	Dark patch mean / std	Bright patch mean / std
HDR fusion	**72.5 / 8.76**	**239.8 / 5.35**

Read the table row by row: the short exposure's dark-patch std of 2.35 is almost equal to the readout noise $\sigma = 2$ — the stripes are completely drowned; the mid exposure's bright patch has mean 253.5 and std 2.03, the residue left after clipping; the long exposure's bright-patch std is exactly 0.00 — total saturation, information reduced to zero. **Of the four images, only the fused one retains clear stripe contrast in both detail patches at once** (std 8.76 and 5.35, both several times the noise) — and that is precisely the value of HDR.

The experiment also throws in a bonus teaching point: the program prints a recovered irradiance peak of $\hat{E}_{\max} = 300.0$, while the scene's ground truth is 272. The error comes from the short-exposure frame — its $\sigma = 2$ readout noise is amplified by a factor of $1/0.25 = 4$ when converted back via v/t; at the brightest pixels, where the mid and long exposures are both saturated, the short exposure is the only "valid vote" left, and its 4×-amplified noise is pushed straight onto the estimate. The reminder: what a short-exposure frame contributes to the irradiance estimate is **amplified noise**, and a more refined implementation would multiply the triangle weight by an additional exposure-time-dependent SNR weight.

12.4 SciVision Implementation

This chapter's SDK usage needs to be disclosed honestly, because it is itself a lesson in engineering. We had originally planned to use `SciSvEdgeEnhance::EnhanceEdges` for sharpening and the `SciSvHDR` family for exposure fusion, but systematic testing while developing this chapter's examples (v3.1) reached the following conclusions:

- A full parameter sweep of `SciSvEdgeEnhance::EnhanceEdges` over `type`/`offset`/`factor`/`brightness` shows that every combination outputs a **binary edge map** of 0/255, not a sharpened "original + enhanced edges" image —

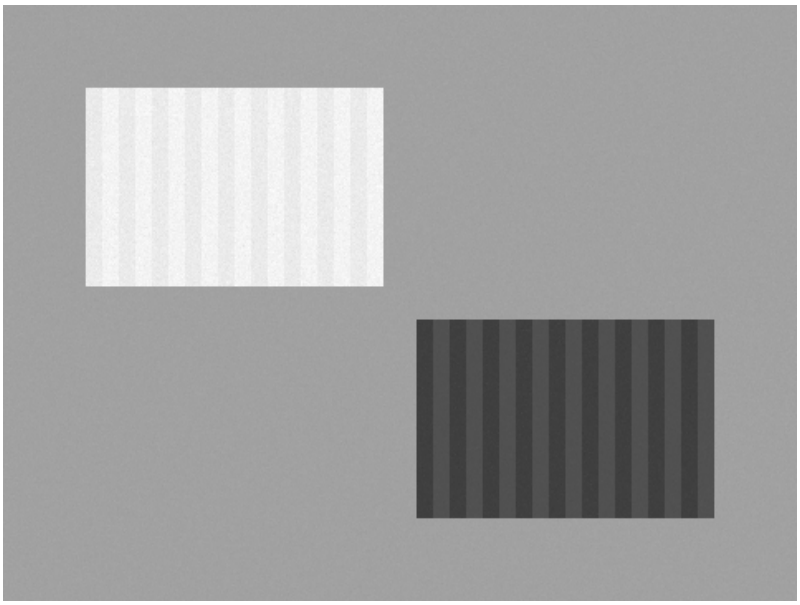


Figure 12.4: The HDR result after fusing the three frames and tone-mapping with $\gamma = 1/2.2$: the bright-region stripes and the dark-region stripes are clearly visible at the same time.

it is actually an edge-map generator, and the name is misleading;

- `SciSvHDR::FusedPicture` returns code 0 for grayscale input but crushes the entire dark region into dead black with a mean of about 0.1 (all detail lost); 3-channel input directly triggers an OpenCV `cv::Mat` assertion crash;
- `SciSvHDR::Compose`, given 480×360 single-channel input, returns a 3-channel image cropped to 449×340;
- `SciSvHDR::ToneMapping(method=1)` triggers an OpenCV `cv::scaleAdd` assertion and terminates the process outright (exit code 0xC0000409).

Therefore both the sharpening and the HDR fusion in this chapter are hand-written, and the SDK handles only the two things it does reliably: Gaussian filtering and image I/O. The Gaussian part is identical to Chapter 6:

```
SCIMV::SciSvFilter flt;
SciImage soft, blur;
flt.Gaussian(imgSharp, roi, &soft, 9, 9, 1.5, 1.5);  // simulate slight defocus
flt.Gaussian(soft, roi, &blur, 9, 9, 1.5, 1.5);      // blur term of the unsharp mask
```

The unsharp-mask core is a single line of arithmetic, executing $\text{out} = \text{in} + \lambda(\text{in} - \text{blur})$ per pixel and clamping to [0,255]:

```
double v = in[i] + lambda * (in[i] - blur[i]);
out[i] = (unsigned char)(v < 0 ? 0 : (v > 255 ? 255 : (int)(v + 0.5)));
```

The implementation leaves pixels within 5 px of the image border untouched: subtracting the SDK Gaussian's extrapolated border values from the original produces spurious differences, and enhancing them directly would frame the image with black-and-white border artifacts. The core of exposure fusion is equally compact — per pixel, accumulate the triangle-weighted votes over the three frames:

```
for (int k = 0; k < 3; ++k) {
    double v = (double)exps[k][i];
    double w = (v < 255.0 - v ? v : 255.0 - v) + 1e-3;  // triangle weight
    sw += w;
```

```
    se += w * v / t[k];      // t[] = {0.25, 1.0, 4.0}
}
E[i] = se / sw;              // then tone-map with gamma=1/2.2
```

The complete project is in `code/advanced_enhancement/`, and the file header preserves the full test record of the SDK defects above. The engineering lesson of this episode deserves bold type: **every API of a commercial vision library must have its actual behavior verified with a golden experiment of known ground truth before it goes onto the line** — construct a synthetic input whose result you can compute by hand, feed it to the API, and check the output. This is of one lineage with the methodology of Chapter 5, where calibration results are checked against a known target: the documentation is a promise; the experiment is the behavior.

Industry Case: HDR Code Reading on Laser-Etched Battery Shells

A battery-shell production line laser-etches batch codes onto polished aluminum shells, with the code-reading camera facing the highly reflective metal surface head-on: in a single exposure, either the character strokes saturate into solid white or the background is crushed into dead black — the contrast between characters and background never lines up, and the read rate hovered around 70%. The retrofit was to capture two exposure levels back-to-back at the same station (short exposure to keep the characters, long exposure to keep the background), fuse them into one frame by this chapter's weighted scheme, and feed that to the code reader; the read rate rose above 99%. The cost was equally clear: one extra frame per product doubled the acquisition cycle time, and the line was forced to slow down. The alternative route evaluated later was to switch to a 12-bit high-dynamic-range camera, covering the full gray-level span in one frame — more expensive hardware, but no cycle-time penalty. The general conclusion of this case: **for dynamic-range problems, think hardware first (deeper bit depth, larger full well), then algorithms (multi-exposure fusion)** — the algorithmic route saves money but pays in time.

12.5 Summary

- **Enhancement is amplification, not restoration.** The unsharp mask $g = f + \lambda(f - G_\sigma * f)$ amplifies the high-frequency content by $1 + \lambda$, compensating the MTF attenuation caused by the lens and defocus, but detail whose MTF has already reached zero is beyond anyone's saving, and the noise gets amplified along the way.
- **Bound λ by the overshoot.** Extract profiles on representative edges and stop once the overshoot amplitude reaches the noise amplitude (in this chapter $\lambda = 1.0$ with overshoot +5 passes; $\lambda = 4.0$ with overshoot +33 is over the line); overshoot shifts caliper edge-finding systematically, and halos manufacture false defects.
- **When dynamic range falls short, make it up with multiple exposures.** Irradiance recovery $\hat{E} = \sum w(v_i)v_i/t_i \,/\, \sum w(v_i)$ with triangle weights — votes from too-dark and too-bright pixels don't count; in the experiment only the fused image made the bright and dark details visible at the same time.
- **The noise of short-exposure frames is amplified by $1/t$** (in this chapter $\hat{E}_{\max} = 300$ against a ground truth of 272); a refined implementation should add an SNR weight, and the engineering cost of fusion is a doubled cycle time — dynamic-range problems should first be weighed against the hardware route of a higher-bit-depth camera.
- **The behavior of commercial-library APIs must be verified with golden experiments.** This chapter's SDK sharpening and HDR interfaces were all either mislabeled or outright crashed; the final solution let the SDK do only Gaussian filtering and I/O, with the core algorithms hand-written — the documentation is a promise; the experiment is the behavior.

The standard treatment of sharpening methods such as the unsharp mask is the textbook by Gonzalez and Woods (Gonzalez and Woods 2018); the methodological origin of this chapter's multi-exposure fusion is the work of Debevec and Malik at SIGGRAPH 1997, which recovers the camera response function and

fuses a high-dynamic-range radiance map (Debevec and Malik 1997), while the complete framework of high-dynamic-range imaging — from acquisition through display to tone mapping — is the monograph by Reinhard et al. (Reinhard et al. 2010). For frequency-domain analysis of sharpening filters and more systematic gray-level transformation methods, see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

Part IV

Locating

This part covers the core algorithms for finding targets in an image: edge detection, detection of geometric primitives, the Hough transform, template matching, shape matching, and feature and color matching, concluding with ROI generation and fixturing on production lines.

13 Edge Detection

The most information-dense places in an industrial image are where the gray value becomes **discontinuous**: the boundary between workpiece and background, the contour of a hole, the strokes of a character, the two sides of a scratch — nearly all the geometric information needed for locating and measurement is carried by these **edges**. The gray value of a flat region drifts with the illumination, but the position of an edge stays comparatively stable — which is why visual measurement places its trust in edges. Chapter 14 solved the one-dimensional problem of "we roughly know where the edge is; localize it precisely along a search line"; this chapter answers the more fundamental question — when we do not know in advance where the edges are, how do we find all of them in the entire image and obtain a two-dimensional **edge map**? The edge map is the standard input to downstream algorithms such as contour analysis, shape matching, and the Hough transform of Chapter 15.

The experiments of this chapter run on a single 480×360 synthetic scene (Figure 13.1): on a background of gray value 80, the upper part holds a pair of high-contrast shapes — a rectangle and a circle at gray value 180 (contrast 100); the lower part holds a pair of low-contrast shapes at gray value 100 (contrast only 20); squeezed between the two columns of shapes is a vertical texture band, a two-dimensional sinusoidal texture of amplitude ±5 and period 12 px (simulating surface textures such as brushed metal or woven fabric); the whole image is overlaid with Gaussian noise of standard deviation $\sigma = 8$ (random seed fixed for reproducibility). Strong edges, weak edges, texture, and noise compete in the same arena — a good edge detector should report the first two and reject the last two.

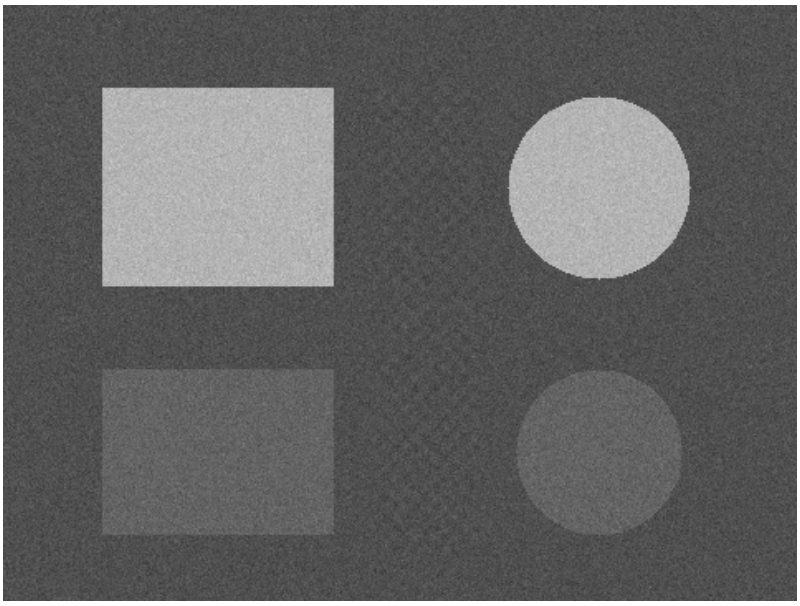


Figure 13.1: The synthetic test scene (480×360): background gray value 80; on top, a high-contrast rectangle and circle (gray value 180, contrast 100); below, a low-contrast rectangle and circle (gray value 100, contrast 20); the vertical band in the middle is a sinusoidal texture of amplitude ±5 and period 12 px; the whole image is overlaid with Gaussian noise of $\sigma = 8$.

13.1 The Gradient and First-Order Operators

An edge is a rapid change of gray value along some direction, and "rate of change" is precisely the derivative. For a two-dimensional image, combining the horizontal and vertical partial derivatives gives the **gradient** $\nabla f = (g_x, g_y)$: its direction points where the gray value rises fastest, and its magnitude measures how violent the change is. A discrete image has no true derivatives; we can only approximate them by differences of neighboring pixels. The most widely used approximation is the **Sobel operator** — a pair of 3×3 convolution kernels:

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

h_x takes a central difference in the horizontal direction and a [1 2 1]-weighted smoothing in the vertical direction; h_y does exactly the opposite. Differencing and smoothing run perpendicular to each other — differentiation itself amplifies noise, so averaging first in the direction perpendicular to the derivative is a way of using the tools of Chapter 6 to "insure" the difference.

The two components yield the gradient's **magnitude** and **direction**:

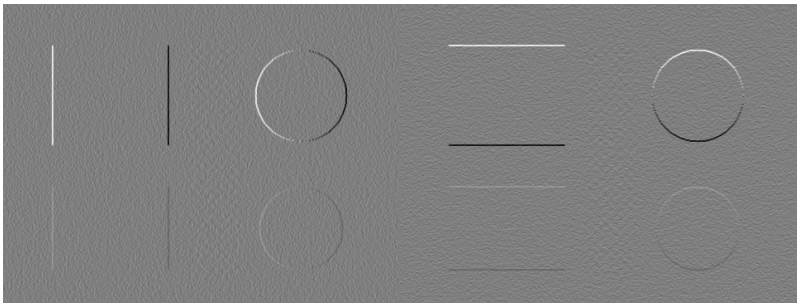
$$A = \sqrt{g_x^2 + g_y^2}, \qquad \theta = \operatorname{atan2}(g_y,\, g_x).$$

The magnitude answers "how edge-like is this," the direction answers "which way does the edge run" (the gradient direction is perpendicular to the edge's course), and the Canny pipeline will need both.

First, observe the **directional selectivity** of the components. Figure 13.2 shows the two signed components of a 3×3 Sobel applied to the test scene (displayed with an offset of 128 and a scale of 1/4: brighter than mid-gray is positive, darker is negative). g_x lights up only the vertical edges: the rectangle's left edge is positive (crossing from 80 into 180, gray value rising) and its right edge negative; the circle's ring is bright on the left

The Sobel kernels are separable: $h_x = [1\ 2\ 1]^{\mathsf{T}} \otimes [-1\ 0\ 1]$, i.e. "vertical smoothing $\times$ horizontal differencing." Replace the smoothing part with [1 1 1] and you get the Prewitt operator; replace it with [3 10 3] and you get the Scharr operator with better rotational symmetry — the skeleton is always "smooth in one direction, difference in the other."

and dark on the right — the same circle, opposite gradient directions on its two sides; the rectangle's top and bottom edges vanish completely in g_x. The behavior of g_y is exactly the transpose: it sees only horizontal edges, positive on top, negative on the bottom. The responses of the low-contrast shapes are only 1/5 of the strong ones and look dim in both images — the edge response is proportional to contrast, and this plain fact is the root of every thresholding problem to come.



(a) Horizontal component g_x (b) Vertical component g_y

Figure 13.2: The signed components of the 3×3 Sobel (displayed with offset 128, scale 1/4). (a) g_x responds only to vertical edges; the rectangle's left edge is positive (bright), its right edge negative (dark), and the circle's ring has opposite signs on its left and right sides; (b) g_y responds only to horizontal edges. The responses of the low-contrast shapes are merely 1/5 of the strong ones.

Combining the two components into the magnitude, the directional information disappears and all edges brighten alike, giving Figure 13.3 — this is the direct output of the SDK's `SobelAmplitude`. The strong contours are crisp and bright, the weak contours faintly discernible, and the background is carpeted with the granular response of noise: differentiation amplified the noise, and with a first-order operator alone we obtain only a grayscale map of "edge likelihood" — still several crucial steps away from a clean edge map.

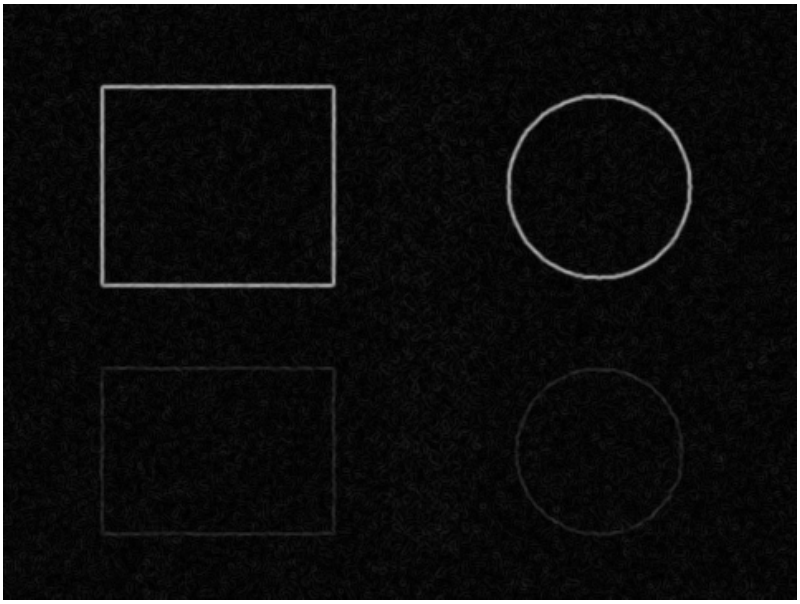


Figure 13.3: Sobel gradient magnitude (SDK `SobelAmplitude`, SUM_SQRT combination). Strong contours bright, weak contours dim, and the background carpeted with the granular response caused by noise.

13.2 A Second-Order Operator: Laplace

The other line of attack is the second derivative. The **Laplacian** is the sum of the second derivatives in the two directions, $\nabla^2 f = \partial^2 f/\partial x^2 + \partial^2 f/\partial y^2$, with the 8-neighborhood discrete kernel

$$h = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Its selling point is localization by **zero crossing**: across a step edge, the first derivative is a single peak, while the second derivative has one extremum on each side of the peak and passes through zero exactly at the edge position — detecting the zero crossings should, in theory, yield naturally closed, single-pixel-wide edge contours with no direction to choose.

The measurements, however, pour cold water on this idea. Figure 13.4 shows the response of the SDK's `Laplace` (3×3, 8-neighborhood kernel, absolute-value output) on the test scene: the entire image is drowned in noise grain, and the strong contours survive only as faintly visible **double lines** — the second derivative has one extremum on each side of the edge, which under absolute-value display becomes two parallel streaks of response flanking the true zero-crossing position in between. The weak contours sink into the noise entirely. Two causes compound: first, second-order differencing amplifies high-frequency noise far more than first-order differencing does, and once the $\sigma = 8$ noise has been churned through it, the useful signal is gone; second, the Laplacian provides no directional information, so downstream processing cannot refine along the edge normal the way the gradient allows. This is why **first-order operators dominate absolutely** in industrial practice — the theoretical elegance of the second-order operator is no match for its fragility to noise.

The classic remedy that makes second-order operators practical is LoG (Laplacian of Gaussian): smooth with a Gaussian first, then take the Laplacian, which is equivalent to a single convolution with a "Mexican hat"-shaped kernel; its difference approximation, DoG (the difference of two Gaussians with different σ), remains alive and well in blob detection and SIFT features.

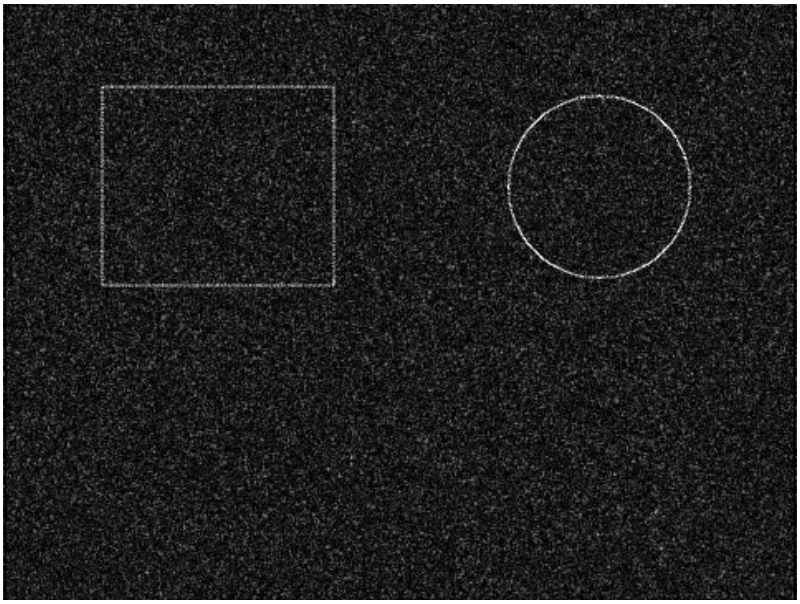


Figure 13.4: Laplace response (3×3, 8-neighborhood kernel, absolute-value output). The noise is fiercely amplified by the second-order differencing; the strong contours retain only a faint double-line response, and the weak contours are completely submerged.

13.3 The Canny Pipeline

The acknowledged standard answer for turning the output of a first-order operator into a clean, single-pixel-wide, connected edge map is the Canny algorithm (Canny 1986). Canny formulated "good edge detection" as three criteria — good detection (no misses, no false alarms), good localization, and single response (one edge reported once) — and derived a near-optimal realization: a four-step pipeline.

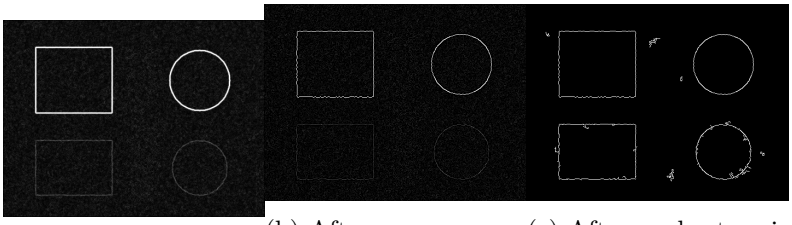
Step 1: Gaussian smoothing. Differentiation amplifies noise, so first press the noise down with a Gaussian filter (here 3×3, $\sigma = 0.8$). σ is the knob of detection scale: larger means more noise-resistant, but neighboring edges get blurred together and localization grows less precise.

Step 2: Gradient computation. Apply Sobel to the smoothed image to obtain the magnitude A and direction θ (Figure 13.5a). Note that the strong contours are now bright bands several pixels wide — smoothing has spread the step out, and the gradient peak has widened with it.

Step 3: Non-maximum suppression (NMS). The gradient magnitude forms a ridge in the direction perpendicular to the edge, and the true edge should lie only on the ridge line. NMS works as follows: quantize each pixel's gradient direction into 4 sectors (horizontal, 45°, vertical, 135°), compare the magnitude with one neighbor on each side along the gradient direction, and keep the center only if it is no smaller than one side **and strictly greater than** the other; otherwise set it to zero. The asymmetry of the comparison ($\geq$ on one side, $>$ on the other) is deliberate: when a flat plateau appears at the ridge top, only one side is kept, guaranteeing that the thinned result is single-pixel. In Figure 13.5b, the bands several pixels wide have been pared down to a thin ridge — but the local maxima of the noise have survived just the same; NMS handles "thin," not "true."

Step 4: Hysteresis thresholding with two thresholds. Separating edges from noise with a single threshold is doomed to sacrifice one side or the other, so Canny uses two: pixels with magnitude $\geq$ the high threshold are **seeds** — strong enough to be trusted as edges; pixels with magnitude between the low

and high thresholds are **candidates** — insufficient evidence on their own, but if 8-connected to a seed, they are accepted as the continuation of an edge. The implementation performs region growing from all seeds, expanding into every connected pixel whose magnitude is $\geq$ the low threshold. The intuition is clear: a weak response hugging a strong edge is most likely the part of the same edge where the contrast fades; an isolated weak response is most likely noise. The strong vouch and the weak follow — only this way can a contour whose contrast waxes and wanes be preserved complete and connected.



(a) Gradient magnitude (b) After non-maximum suppression (c) After hysteresis thresholding (20, 60)

Figure 13.5: Step-by-step results of the Canny pipeline. (a) Gradient magnitude after Gaussian smoothing; the strong contours are bright bands several pixels wide; (b) NMS pares the bands down to single-pixel thin ridges, with the noise maxima still present; (c) hysteresis thresholding (low 20, high 60) outputs a clean, connected, single-pixel edge map in which the contours of all four shapes are preserved.

The final result, Figure 13.5c, contains **2163** edge pixels: the contours of all four shapes are **fully preserved**, single-pixel wide, and almost completely connected, with even the low-contrast rectangle and circle closing up intact. As a control, skipping NMS and hysteresis linking and binarizing the gradient magnitude directly with a single threshold (the same value, 60) yields Figure 13.6: **4064** edge pixels, nearly twice as many. Every extra pixel is waste — the strong contours become coarse bands 2–4 px thick (the gradient peak has width, and a single threshold scoops in the entire peak top), while the weak contours shatter into broken dashed lines (the magnitude oscillates

about the threshold; what crosses stays, what doesn't breaks). An edge map of uneven thickness and intermittent gaps is terrible input for downstream contour tracing and geometric fitting; NMS cures the "coarse," hysteresis linking cures the "broken" — and that is precisely the respective value of Canny's last two steps.

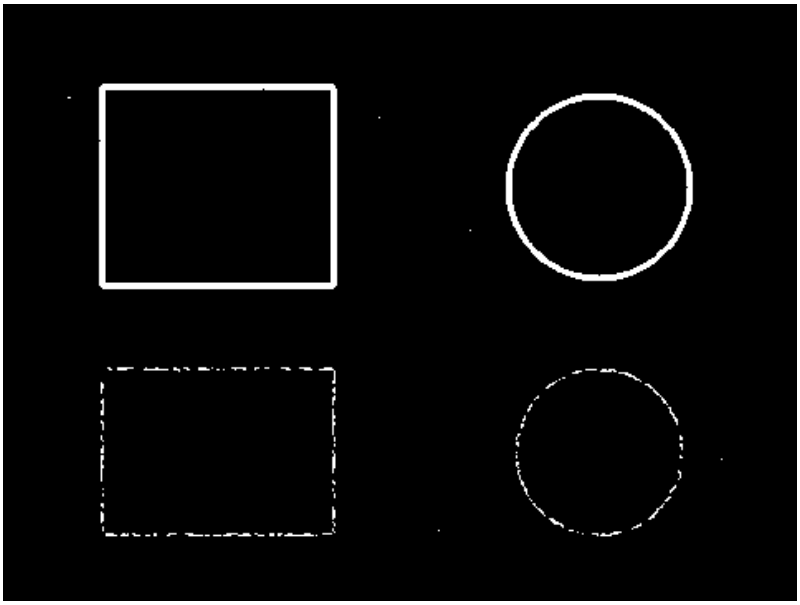


Figure 13.6: Control: binarizing the gradient magnitude directly with a single threshold (threshold 60), 4064 px. Strong contours grow as thick as 2–4 px, and weak contours shatter into dashed lines — compare with the 2163 px single-pixel connected result of Figure 13.5c.

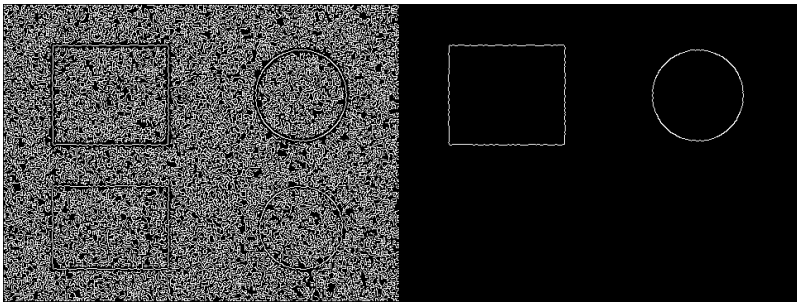
13.4 Choosing the Thresholds in Practice

Canny turned one threshold into two, and the responsibility of choosing them doubled as well. Both directions of failure are worth seeing with your own eyes (Figure 13.7).

Lowering the threshold pair to (8, 24) yields Figure 13.7a: **64077** edge pixels — 30 times the correct result. The gradient

responses of the noise cross the low threshold en masse, then chain to one another through hysteresis linking, turning the whole image into a dense web; the sinusoidal grid of the central texture band also materializes in full. There is a design detail here worth spelling out: this scene's texture has amplitude ± 5 and period 12 px, and its gradient-magnitude peak measures about 19 — pressed just below the default low threshold of 20. That is why the texture region is spotless in the (20, 60) result, while under (8, 24) it floods out together with the noise. On real production lines, brushed, sandblasted, and woven surfaces all play the role of this texture band: whether they count as "edges" depends entirely on where you draw your threshold.

In the opposite direction, raising the threshold pair to (40, 120) yields Figure 13.7b: only **918** edge pixels remain, and the image is immaculately clean — but the low-contrast rectangle and circle have **vanished entirely**. After smoothing, the gradient magnitude of a contrast-20 edge falls short of 120; not a single seed can be produced, and hysteresis linking cannot make bricks without straw. In detection tasks, a miss is usually deadlier than a false alarm: false alarms can still be salvaged by downstream screening, but a missed target never comes back.



(a) Thresholds too low (8, 24): 64077 px

(b) Thresholds too high (40, 120): 918 px

Figure 13.7: Threshold-sensitivity experiment. (a) Low threshold pair: noise and texture cross the line en masse and chain together, drowning the true contours in a dense 64077 px web; (b) high threshold pair: the image is clean, but the weak contrast-20 rectangle and circle are lost completely, leaving only 918 px.

The engineering rule of thumb boils down to two steps. **Step one, fix the high threshold: take 3–4 times the noise gradient level.** Measure the typical level of the gradient magnitude in a blank region of the image (e.g. mean plus three standard deviations, or a high quantile) and anchor the high threshold at 3–4 times it — this guarantees that seeds almost cannot be produced by noise. **Step two, fix the low threshold: start from 1/2 to 1/3 of the high threshold.** The low threshold governs contour connectivity: too close to the high threshold and hysteresis linking exists in name only; too low and the noise builds bridges. This chapter's (20, 60) was chosen exactly by this rule: after smoothing, the $\sigma = 8$ noise has a gradient magnitude of roughly 15, the high threshold of 60 is about 4 times that, and the low threshold is 1/3 of the high. Finally, fine-tune online with real images — the rule gives the starting point, not the destination.

13.5 Subpixel and Outlook

The edge pixels Canny outputs have only integer coordinates, nailing the localization precision to 1 px. For recognition and counting this is enough; for measurement it is nowhere near — in a system at 10 m/px, 1 px of quantization error is 10 m. Fortunately the remedy is cheap: NMS compares precisely the three magnitudes along the gradient direction, g_{-1}, g_0, g_{+1}, and a parabolic interpolation over them pins the ridge-top position down to **subpixel** accuracy — the very same vertex formula that Chapter 14 used on one-dimensional profiles. Interpolating point by point along the contour upgrades the pixel chain to a subpixel contour. In industrial practice, if only a local edge or hole needs measuring, the more common tool is the caliper of Chapter 20, doing one-dimensional subpixel localization directly inside an ROI; whole-image edge detection is for the situations where you do not know in advance where the target is.

The SciVision SDK also offers a class of subpixel extractors complementary to Canny: `SciSvExtractLinesGauss`, which implements Steger-style curvilinear-structure detection (Steger

This is kin in spirit to the histogram-based threshold selection of Chapter 7: there we thresholded the **gray-value** distribution to separate foreground from background; here we threshold the **gradient-magnitude** distribution to separate edges from noise. The object changed; the methodology of "look at the distribution, find the divide" did not.

1998). Note that what it finds are not step edges but **lines** — ridges or valleys with width, such as scratches, bonding wires, or adhesive tracks. Based on Gaussian derivatives, `ExtractLinesGauss` directly outputs an array of center-line contours at subpixel accuracy, attaching attributes to each line: direction `angle`, response strength `response`, left/right half-widths, asymmetry, and contrast. Its parameters carry familiar shadows: `sigma` is the detection scale, and `lowThresh`/`highThresh` are the hysteresis threshold pair — the same idea as in Canny; `lineModel` selects among three profile models: Gaussian line, bar line, and parabolic. The companion `CalculateParameter` can even derive `sigma` and the two thresholds from the expected line width and contrast, sparing manual tuning. This chapter ran no experiment on it and offers it only as a pointer: when your target is "a line with width" rather than "a gray-value step," this is the tool to remember.

13.6 SciVision Implementation

The operators used in this chapter are provided by `SCIMV::SciSvFilter`:

```
SCIMV::SciSvFilter f;
SciImage amp, lap, smooth;
SciROI roi;
SciPoint tl(0, 0), br(W, H);     // GenRect1 bottom-right is an exclusive endpoint: pass (W, H)
roi.GenRect1(tl, br);

// Sobel gradient magnitude: sobelType=SUM_SQRT(1) is sqrt(gx^2+gy^2); kernelSize=3 (size of t
long rc = f.SobelAmplitude(img, roi, &amp, SCIMV::SUM_SQRT, 3);
// Laplace: 3x3 kernel, filterMask=1 (8-neighborhood kernel), resultType=0 (absolute-value out
rc = f.Laplace(img, roi, &lap, 3, 1, 0);
// Canny pre-smoothing: 3x3 Gaussian, sigma=0.8
rc = f.Gaussian(img, roi, &smooth, 3, 3, 0.8, 0.8);
```

`SobelAmplitude`'s `sobelType` selects how the magnitude is combined, `SUM_SQRT` being the standard square root of the sum of squares; `kernelSize=3` is the size of the **internal**

Gaussian pre-smoothing kernel (valid range [1,11]) and has nothing to do with the size of the Sobel derivative kernel. `Laplace`'s `filterMask=1` selects the 8-neighborhood kernel (center −8), and `resultType=0` outputs the absolute value — the double response of Figure 13.4 stems precisely from this.

The SDK also has a one-shot `Canny(srcImage, ROI, dstImage)` — but it has **no threshold parameters whatsoever**; the low and high thresholds are entirely built in, cannot be tuned per scene, and naturally cannot perform the sensitivity experiment of Section 13.4. This chapter therefore hand-wrote NMS and hysteresis linking (full code in the example project) — which happens to be the part most worth writing once with your own hands. If production code needs a Canny with controllable thresholds, the same advice applies: own these two steps yourself, leave the gradient and the smoothing to the SDK, and keep the decision logic in your own hands.

There is also a pitfall that must be recorded honestly: in the current version, `SciSvFilter::Gaussian` corrupts the rightmost 1–2 columns of the output image no matter what ROI is passed (with a kernel of 3, the last 2 columns are zeroed). The zeroed columns form a false step of contrast above 80 against their neighbors, showing up in the edge map as a false edge running the full image height. The example project's countermeasure is to repair by replicating the last valid column to the right:

```
// SDK defect workaround: Gaussian zeroes the rightmost 2 output columns; fill them by copying
for (int r = 0; r < H; ++r) {
    smooth[r * W + W - 2] = smooth[r * W + W - 3];
    smooth[r * W + W - 1] = smooth[r * W + W - 3];
}
```

The complete runnable project that generates all of this chapter's images and statistics is located at `code/edge_detection/`; all edge-pixel counts (4064 / 2163 / 64077 / 918) are actual program output.

Industry Case: Threshold Tuning for Glass Edge-Chip Detection

A glass deep-processing line used Canny to extract the edge contour, then inspected the contour for local gaps to flag edge chips. The low and high thresholds were tuned against the images available at installation acceptance and then frozen. Months later, the night shift's miss rate climbed: LED lumen decay compounded by lower ambient light at night reduced image contrast, the gradient magnitudes of small, faint chips fell below the frozen high threshold, and no seeds could be produced. The fix was per-shift automatic tuning: at the start of each shift, estimate the noise gradient level in a blank region beside the edge (a high quantile of the gradient magnitude), set the high threshold $= K\times$ that estimate ($K \approx 3.5$), and the low threshold to half the high. Through several rounds of lamp aging and replacement since, the miss rate has held steady. The lesson: a gradient threshold measures "how much stronger the signal is than the noise"; a fixed absolute value silently decays along with the illumination — anchor the threshold to the noise, not to one day's good lighting.

13.7 Summary

- **Edges are gray-value discontinuities, characterized by the gradient**: Sobel approximates the gradient with a pair of 3×3 kernels that "difference in one direction, smooth in the perpendicular one"; the magnitude decides "how strong," the direction decides "which way." The edge response is proportional to contrast — the root of every thresholding problem.
- **Second-order operators are theoretically elegant, practically fragile**: under $\sigma = 8$ noise, Laplace's zero-crossing localization degenerates into a screenful of grain plus double-line responses, and it provides no directional information — industrial edge detection is dominated by first-order operators.
- **Canny's four steps each do one job**: Gaussian smoothing resists noise, the gradient computes "likelihood," NMS pares the response band down to a single-pixel thin ridge, and hysteresis thresholding — the strong vouch, the weak follow — preserves connected

contours whose contrast waxes and wanes. In the experiments, single-threshold binarization at the same threshold output 4064 px of coarse, broken edges, while Canny output 2163 px of single-pixel connected contours.

- **Anchor the thresholds to the noise**: set the high threshold at 3–4 times the noise gradient level, start the low threshold at 1/2–1/3 of the high, then fine-tune online. In this chapter's experiments, (8, 24) let noise and texture drown the result (64077 px), and (40, 120) wiped out the weak contours entirely (918 px).
- **Measurement demands subpixel**: parabolic interpolation over NMS's three magnitudes yields subpixel edges; for line-like structures (scratches, bonding wires), the Steger-style `ExtractLinesGauss` directly extracts subpixel center lines together with attributes such as width and contrast.

Two foundational papers reward a return to the originals: Canny's classic derivation of this chapter's four-step method from optimality criteria (Canny 1986), and Marr and Hildreth's work laying the theoretical basis for second-order zero-crossing (LoG) detection (Marr and Hildreth 1980). For a systematic treatment of subpixel edge and line extraction, detection-scale selection, and their accuracy analysis, see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

14 Detecting Geometric Primitives

Place a part under the camera, and the first class of questions industrial vision must answer is often surprisingly plain: where is this edge? Where is the center of this hole? Alignment and placement close their motion loops on edge positions, dimensional inspection measures the distance between two edges, and robotic pick-up needs hole-center coordinates — in the end they all reduce to the same thing: **locating points, lines, circles, and other geometric primitives in the image with subpixel accuracy**. "Subpixel" is not a marketing word: in an imaging system at 10 m/pixel, if localization is only accurate to whole pixels, the measurement resolution is nailed down at 10 m; proper subpixel localization can reliably reach 1/10 pixel or better — equivalent to improving the system's accuracy by an order of magnitude without changing a single piece of hardware.

This chapter runs all of its experiments on one real industrial image (Figure 14.1): an oblong metal connector part on a dark background (2448×2048 single-channel grayscale, taken from Smart3's standard "find point / find line / find circle" solution). The part's outer contour has a roughly horizontal straight edge along the top — flat in the middle, rounding off at both ends — and inside it are two left-right symmetric bored holes, each with a circular bore mouth exposing the dark background at its center and a bright annular step around it. We will first extract a string of edge points along the top straight edge and fit a line to them, then cover each hole with a circular ROI and fit a circle — the rounded shoulders at the two ends of the straight edge will play the troublemakers, forcing out the full value of robust fitting.

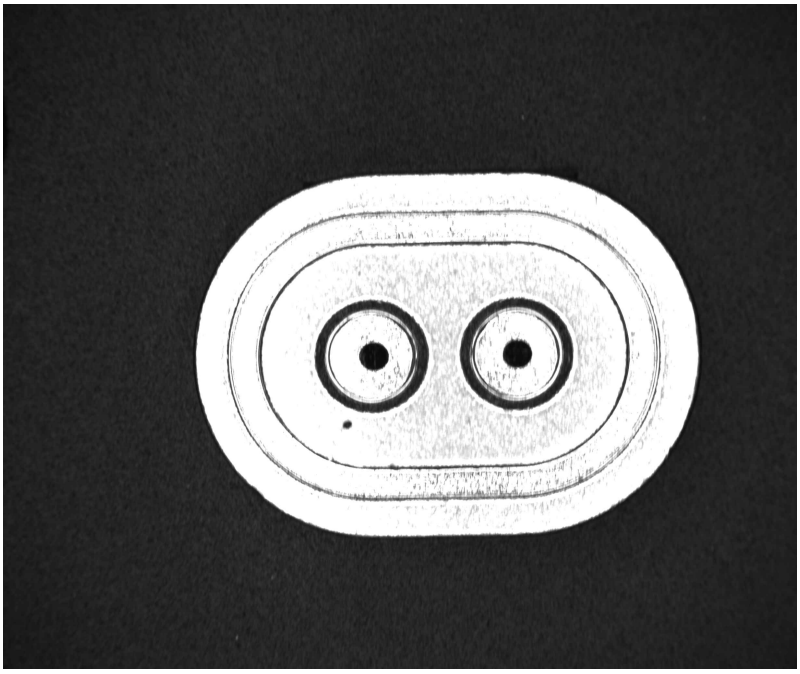


Figure 14.1: The real test image: an oblong metal connector part on a dark background. The outer contour's top edge is a roughly horizontal straight edge, flat in the middle and rounding off at both ends; inside the part are two left-right symmetric bored holes whose centers serve mating alignment and pick-up.

14.1 Extracting Edge Points

To locate an edge, you do not need to run edge detection over the whole image (that is the subject of Chapter 13). The industrial approach is far more direct: inside an ROI that roughly frames the target edge, lay down N mutually parallel **search lines**, splitting the two-dimensional localization problem into N one-dimensional ones. The processing along every search line is identical:

1. **Take a gray profile**: read the gray values point by point along the search line, yielding a one-dimensional curve.
2. **Projection averaging**: average over several pixels in the direction perpendicular to the search line (the projection within the projection width), using the average to suppress single-pixel noise — the same reasoning as in Chapter 6: independent zero-mean noise cancels under averaging.
3. **First-derivative localization**: take the first derivative of the averaged profile; an edge corresponds to an extremum of the derivative magnitude. Extrema whose magnitude falls below a threshold are treated as noise and not reported.
4. **Subpixel interpolation**: let the magnitudes at the discrete extremum of the derivative and at its left and right neighbors be g_0, g_{-1}, g_{+1}; fit a parabola through these three points, and the offset of its vertex relative to the extremum is

$$\delta = \frac{g_{-1} - g_{+1}}{2\,(g_{-1} - 2g_0 + g_{+1})},$$

δ falls within $\pm 1/2$ pixel, and the subpixel edge position is "the whole-pixel extremum position + δ". Three samples determine one parabola, the vertex formula is closed-form, and the computational cost is negligible — subpixel accuracy is almost free.

This machinery comes with a set of parameters that can only be configured well if you understand the principle. The **edge strength threshold** (e.g. `strengthThresh`) is the admission gate on the first-derivative magnitude: set it too low and noise

wiggles get counted as edges; too high, and genuine low-contrast edges are missed. **Polarity** specifies which kind of gray transition is accepted — black→white (rising), white→black (falling), or both: in this example we scan from top to bottom, crossing from the dark background into the bright part, a textbook black→white edge; locking the polarity immediately filters out half of the false edges. The **edge width** is the expected width of the gray transition band and determines the differencing span of the derivative: too small and it is noise-sensitive, too large and two adjacent edges get smeared into one. The **projection width** is a trade-off between noise suppression and blurring: the wider the projection, the more noise is averaged away and the more stable the edge strength estimate; but if the edge is not strictly perpendicular to the search line, too wide a projection "smears" the tilted edge sideways and the localization actually shifts — in practice 3 to 7 pixels is most common. The **edge type** decides which edge to report when a search line finds multiple candidates: the first, the last, or the one with the best strength; "best" is the most robust against weak false edges caused by oil stains and reflections.

Figure 14.2 shows the extraction result with 60 search lines laid across the top-edge ROI: the ROI deliberately overruns the flat segment and reaches into the rounded shoulders at both ends, so among the 60 white crosses — the ones in the middle line up neatly along the true straight edge, while the ones at the two ends shift downward following the rounded corners. The search lines faithfully report the edge position they "see", and that is how the shoulder points sneak into the point set (18 of them, by the criterion of distance $|d| > 5$ px to the fitted line). How not to be dragged off course by them is the subject of the next section.

14.2 Line Fitting: Least Squares and Robust Methods

With a string of edge points in hand, the next step is to estimate a line from them. **Least squares (LSQ)** fitting minimizes the sum of squared perpendicular distances from all points to the line

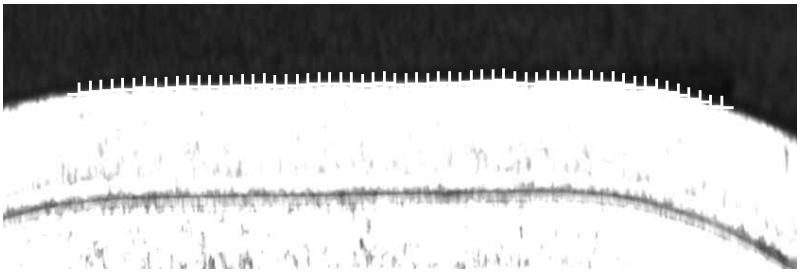


Figure 14.2: Edge points extracted along the top by 60 search lines (white crosses). The middle ones line up along the true straight edge, while the points that overrun into the rounded shoulders shift downward following the contour, becoming the outliers in the subsequent fit.

$$\min_{L} \sum_{i} r_i^2, \qquad r_i = \mathrm{dist}(\mathbf{p}_i,\, L),$$

and the solution is closed-form (the line passes through the centroid of the point set, with its direction given by the principal eigenvector of the scatter matrix) — fast and free of hyperparameters. When all points are "roughly correct" it is optimal — the same statistical fact that makes averaging optimal under Gaussian noise. But it is extremely sensitive to **outliers**, and the reason hides in that square: residuals are squared before they are summed, so a shoulder point off by 30 px contributes 3600 times as much to the objective as a normal point off by 0.5 px. To appease a handful of large-residual points, the fitted line would rather rotate as a whole and let the majority of good points each be a little more wrong — the squared penalty being dominated by large residuals is the root cause of LSQ's instability.

The idea of **robust estimation** is to replace the squared penalty with one that is more forgiving of large residuals. Huber's scheme (Huber 1964) is equivalent to giving each point a weight that varies with its residual:

$$w(r) = \begin{cases} 1 & |r| \le \kappa \\ \kappa/|r| & |r| > \kappa \end{cases}$$

Points whose residual does not exceed the tuning constant κ enjoy a full say; points beyond it have their weight decay as $\kappa/|r|$ — the larger the residual, the smaller the voice, but never a one-vote veto. Since the weights depend on the residuals and the residuals depend on the fit, the solution uses **iteratively reweighted least squares (IRLS)**:

```
Input: edge point set {p_i}, tuning constant
1. Fit an initial line by ordinary least squares, all weights w_i ← 1
2. Repeat until convergence (line parameters change little enough,
   or the maximum number of iterations is reached):
   a. Compute each point's distance r_i to the current line
   b. Update weights by the Huber rule: if |r_i|    then w_i ← 1,
      otherwise w_i ←  /|r_i|
   c. Do a weighted least-squares fit with weights w_i, updating the line
3. Output the line and each point's final residual
```

In each iteration the outliers' weights are pushed further down and the line steps closer to the "majority of good points"; convergence typically takes three to five rounds.

Talk is cheap, so let us run a controlled experiment on the 60 edge points from the previous section. The ROI overruns a stretch of rounded shoulder at each end, and the search lines falling into those stretches produce a small clutch of outliers shifted downward as a group (18 in total, three tenths of the 60 points). On the LSQ side, outlier rejection is deliberately switched off (`rejectRatio=1`, `rejectDist=20` — a rejection threshold of 20 px lets every shoulder outlier participate in the fit); the Huber side uses `rejectRatio=10`, `rejectDist=5`. We evaluate both fitted lines under one uniform rule — the root-mean-square error (RMSE) over only the inliers whose distance to the respective fitted line satisfies $|d| \le 5$ px: **2.793 px for LSQ, 1.456 px for Huber**, nearly a twofold difference. Geometrically, the LSQ line is dragged down as a whole by the two shoulders and tilted by their asymmetric distribution, while the Huber line hugs the middle straight edge firmly.

The other famous robust route is RANSAC (random sample consensus) (Fischler and Bolles 1981): repeatedly draw a minimal sample at random (a line needs only 2 points), fit it, count the "consensus" points falling within a tolerance band, and keep the model with the largest consensus. IRLS starts from all the points and converges by **reweighting**; RANSAC relies on **random sampling** to stumble upon an outlier-free sample. When the outlier fraction is very high (approaching or even exceeding half), RANSAC is more reliable; when outliers are few and the normal points carry continuous noise, IRLS is more accurate, faster, and its result is repeatable (no randomness) — industrial edge fitting mostly belongs to the latter case.

14.3 Fitting Circles and Ellipses

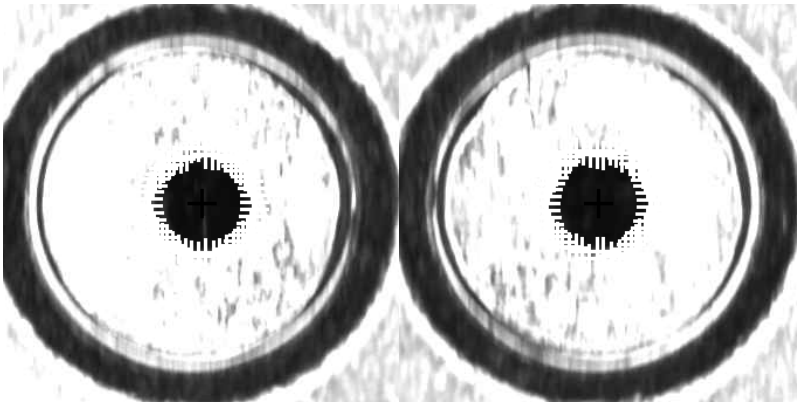
When locating a circular hole, the search lines switch from a parallel layout to a **radial layout**: cover the hole with a circular ROI and lay out search lines uniformly over angle, each scanning along the radial direction (from inside outward); everything else — projection averaging, first derivative, subpixel interpolation — is exactly the same as in the line case. The radial layout adds two key parameters: the **expected radius** `expectRadius` and the **radius tolerance** `radiusRange`, which constrain the search to an annular search band [expectRadius − radiusRange, expectRadius + radiusRange]. In this example `expectRadius=45` and `radiusRange=30`, giving a search band of 15–75 px. Constraining the search band wins twice: the bore mouth inside the hole and the annular step outside it are both excluded, so false edges drop sharply; and each search line's scan length shrinks accordingly, saving time.

A common evaluation trap: the RMSE returned by an SDK's fitting interface is the **all-point** RMSE — 5.908 px on the LSQ side of this experiment, the bulk of which is the shoulder outliers' own residuals. That number is unsuitable for comparing the accuracy of two fits: outliers have large residuals against any "correct" line, so all-point RMSE punishes the data, not the fit. A fair comparison must use a uniform inlier rule (such as $|d| \le 5$ px in this example).

The experiment runs the circle finder once per hole: all 48 radial search lines hit the hole rim, yielding 48 edge points per hole (the white crosses in the two subfigures of Figure 14.3), with a black cross marking the fitted center. The left hole fits to **center (1143.83, 1085.32), radius 44.60 px**, the right hole to **center (1586.00, 1077.59), radius 44.93 px**, and the inter-hole center distance (pitch) is **442.06 px**. All 48 edge points of each hole are effective points, with zero rejections — the part's hole rims are intact, with no point deviating from the reference circle beyond the rejection threshold. The hole-center coordinates and the pitch are exactly the core quantities needed for connector mating and robotic pick-up: the pin-hole positions fix the pick target, the pitch locks the mating orientation.

Circle fitting comes in two families, **algebraic fitting** and **geometric fitting**: the former minimizes the algebraic residual of the circle equation — closed-form and extremely fast, but its implicit weighting introduces bias; the latter minimizes the true Euclidean distance from each point to the circle — iterative but unbiased. The common practice in industrial libraries is to use the algebraic solution as the initial value and finish with geometric iteration, getting both speed and accuracy.

The robust fit hides one more "by-product" worth pointing out here: **while delivering the reference circle, it also hands over a rejection list of "which points refuse to obey the reference."** On this part the hole rims are intact and the



(a) Left hole: 48 edge points (white crosses) + fitted center (black cross)

(b) Right hole: 48 edge points (white crosses) + fitted center (black cross)

Figure 14.3: Radial edge-point extraction and circle fitting on the two bored holes. (a) the left hole and (b) the right hole each have their rim edge points extracted by 48 radial search lines, and the robust fit gives the center (black cross) and radius; both holes have zero rejected points, their rims intact.

rejection list is empty; but the moment a rim develops a protruding burr, a chip-out, or boring-tool chatter marks, the corresponding edge points deviate from the reference circle and get screened out by the rejection threshold, and the position and clustering of the rejected points become markers of the defect directly. In other words, taking the fitted circle as the nominal contour and checking each edge point's radial deviation, or the distribution of rejected points, accomplishes measurement and defect judgment in one and the same fit. Chapter 26 will develop this idea systematically.

14.4 SciVision Implementation

Edge-point extraction and line fitting are provided by `SCIMV::SciSvLineLocator`:

```
SCIMV::SciSvLineLocator lineLoc;
SciPointArray edgePts;
long rc = lineLoc.LinePointsLocator(src, lineROI, emptyRegion,
    /*strengthThresh*/ 25, /*direction top->bottom*/ 0, /*polarity black->white*/ 0,
    /*edgeWidth*/ 3, /*projectWidth*/ 5, /*edgeType best*/ 2,
    /*searchLineCount*/ 60, /*findPointType first derivative*/ 1, &edgePts);
rc = lineLoc.FitLine(edgePts, /*fitMethod Huber*/ 1, 10, 5, 0, 0, &lineHuber, &rmseHuber, &dist
```

The parameters of `LinePointsLocator` map one-to-one onto the principles of Section 14.1: `strengthThresh=25` is the gate on the first-derivative magnitude; `direction=0` specifies scanning from top to bottom; `polarity=0` accepts only black→white transitions (here we enter the bright part from the dark background); `edgeWidth=3` and `projectWidth=5` are the differencing span and the projection averaging width; `edgeType=2` takes the candidate with the best strength; `searchLineCount=60` search lines; `findPointType=1` localizes by the first-derivative extremum. The output `edgePts` is the 60 subpixel edge points. In `FitLine`, `fitMethod` is 0 for least squares and 1 for Huber, and the following `rejectRatio` and `rejectDist` control outlier rejection during the fit (the number of rejection rounds and the distance threshold) — in

the experiment of Section 14.2, the LSQ side used exactly the pair `1, 20` to effectively switch rejection off.

Circle finding is done in one step (search + fit) by `SCIMV::SciSvEllipseLocator`, called once per hole, reading the center and radius out of the output shape `circleShape`:

```
SCIMV::SciSvEllipseLocator circleLoc;
SciROI circleROI; SciPoint center(holeCx, holeCy);
circleROI.GenCircle(center, 80);   // circular ROI covering the rim search band
SciOverlay circleShape;
SciPointArray circlePts, circleEffect, circleReject;
rc = circleLoc.EllipseLocator(src, circleROI, emptyRegion, /*isEllipse*/ false,
    /*strengthThresh*/ 25, /*direction inside->outside*/ 0, /*polarity black->white*/ 0,
    /*edgeWidth*/ 3, /*projectWidth*/ 5, /*edgeType best*/ 2, /*fitMethod LSQ*/ 0,
    /*searchLineCount*/ 48, /*rejectRatio*/ 10, /*rejectDist*/ 5,
    /*expectRadius*/ 45, /*radiusRange*/ 30,
    &circleShape, &circlePts, &circleEffect, &circleReject);
SciPoint fc; double fr = 0; circleShape.GetCircle(&fc, &fr);   // hole center and radius
```

`isEllipse=false` means fitting a circle rather than an ellipse; `direction=0` makes the search lines scan from the hole center outward; `polarity=0` accepts only black→white transitions (crossing from the dark bore mouth into the bright step); `fitMethod=0` is least-squares fitting, and `rejectRatio=10` controls its outlier-rejection rounds; `expectRadius=45` and `radiusRange=30` delimit the 15–75 px annular search band (the circular ROI radius in this example is correspondingly set to 80). The four outputs are, in order: the fitted circle shape `circleShape`, all edge points `circlePts`, the effective points that participated in the fit `circleEffect`, and the rejected points `circleReject` — the two subfigures of Figure 14.3 plot each hole's `circlePts` and fitted center. `GetCircle` extracts the hole's center coordinates and radius from `circleShape`.

An honest disclosure: in the current SDK version, when `fitMethod=1` (Huber), the RMSE output parameter of `FitLine` returns an invalid value (NaN). The companion project therefore uniformly computes the inlier RMSE itself from the array of point-to-line distances (the last output parameter of `FitLine`) — that is where the two numbers 2.793/1.456

in Section 14.2 come from. Commercial SDKs occasionally have such blemishes, and the engineering countermeasure is: recompute key metrics from the raw outputs (points, distances) wherever possible, instead of trusting summary values unconditionally. The complete runnable project that generates all of this chapter's experimental images and numbers lives in `code/geometric_primitives/`, with its sample image self-contained at `code/geometric_primitives/sample/connector.jpg`.

Industry Case: Battery Electrode Edge Locating

The winding process in lithium battery manufacturing requires the anode and cathode electrode sheets to align with the separator to the 0.1 mm level, and the alignment loop depends on real-time locating of the electrode sheet edges. The difficulty is that slit electrode sheets almost universally carry burrs along their edges: with ordinary least-squares fitting, the burr points drag the edge line off, and the winding alignment inherits a systematic bias — robust fitting such as Huber (or piecewise local fits followed by a median) is mandatory to keep it down. The number of search lines is a direct trade-off between accuracy and cycle time: more lines, higher statistical accuracy of the fit, and linearly more time; production lines commonly use 30–100 lines, taking the upper end as the cycle-time margin allows. Another field-proven point is setting `edgeType` to "best edge": oil stains on the electrode surface and reflections off the rollers create weaker false edges, so choosing "the first edge" easily locks onto the wrong target, while choosing the one with the best strength is essentially immune.

14.5 Summary

- **The standard industrial localization pipeline is "search-line sampling → 1D edge localization → subpixel interpolation → geometric fitting"**: split the 2D problem into N 1D problems; on each search line, projection averaging suppresses noise, the first-derivative extremum pins the edge, and parabolic vertex interpolation yields the subpixel position.

- **Subpixel is almost free, but with preconditions**: parabolic interpolation only delivers its accuracy when the profile is clean enough; the projection width (denoising vs. blurring), the edge strength threshold, the polarity, and the edge type must all be configured for the scene rather than left at defaults.
- **The root of least squares' outlier sensitivity is the squared penalty**: large residuals dominate the objective, and a handful of shoulder outliers is enough to drag the whole line off. Huber weighting + IRLS down-weights large-residual points by $\kappa/|r|$; in this chapter's experiment the inlier RMSE improved from 2.793 px to 1.456 px.
- **Comparing fitting accuracy requires a uniform inlier rule**: all-point RMSE charges the outliers' own residuals to the fit's account — a common evaluation trap.
- **Robust circle fitting delivers measurement and defect clues in one pass**: this chapter fits the center and radius of each of the two bored holes (a pitch of 442.06 px, directly usable for mating and pick-up), with zero rejected points on the intact rims; the moment a rim develops a defect, the rejection list becomes a position marker of it — the reference geometry serves measurement, the outlier list serves defect judgment.

For more accurate direct fitting of circles and ellipses, two classics on the algebraic solution are worth consulting: Taubin's estimation of implicit curves and surfaces (Taubin 1991), and Fitzgibbon et al.'s ellipse-specific direct least-squares method (Fitzgibbon, Pilu, and Fisher 1999). For a more systematic treatment of subpixel edge extraction and geometric primitive fitting (including ellipses, circular arcs, and uncertainty analysis), see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

15 The Hough Transform

The search-line approach of Chapter 14 carries a self-evident premise: you roughly know where the target is. Frame an ROI first, then lay down the search lines — the whole pipeline rests on the prior of "one ROI, one primitive". But what if that premise fails? The image contains **multiple** lines and **multiple** circles, with no advance knowledge of where each one sits; the targets may also be broken (dashed lines, gaps carved out by reflections) or partially occluded. Now there is nowhere to lay search lines, and point-by-point tracking snaps at the first gap. The **Hough transform** asks the question differently: instead of asking "where is this line", it lets **every edge point vote for all the lines that could pass through it (voting)**, and the parameter combination with the most votes naturally wins. This chapter runs the voting machinery end to end on a 480×360 synthetic scene (Figure 15.1): the scene contains 3 lines — one of them a **dashed line with a gap ratio of about 47%** — and 2 circles — one of them **40% occluded** — plus 40 stray blobs and Gaussian noise with =8. Breaks, occlusion, clutter, and noise, all four kinds of interference present at once — exactly what it takes to force out the full value of the Hough transform.

15.1 Parameter Space and Voting

To vote for lines, the lines first need a suitable set of parameters. The Hough transform adopts the **normal form** parameterization (Duda and Hart 1972):

$$\rho = x\cos\theta + y\sin\theta,$$

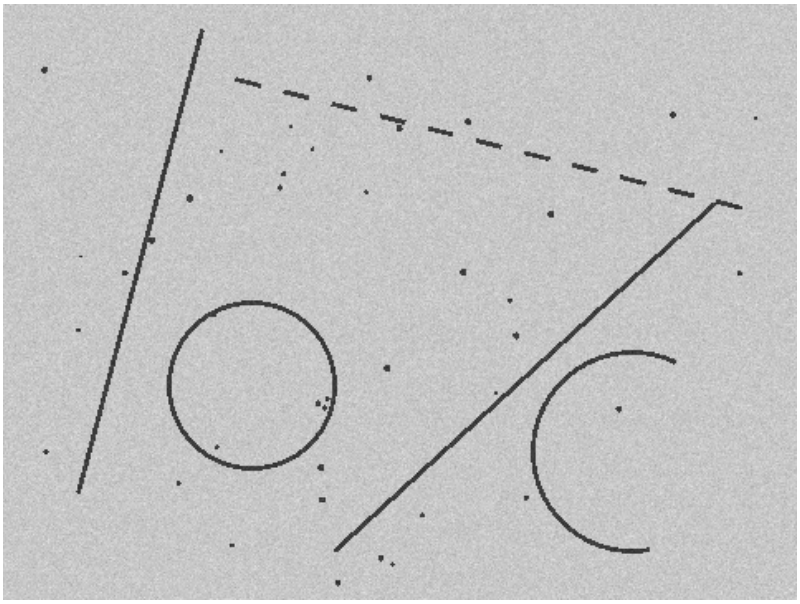


Figure 15.1: The synthetic test scene (480×360): 3 lines (L1 and L3 solid, L2 a dashed line with a gap ratio of about 47%), 2 circles (C1 complete, C2 missing 40% of its arc), and 40 stray blobs, overlaid with Gaussian noise of =8.

where $\theta \in [0^\circ, 180^\circ)$ is the angle between the line's normal direction and the x axis, and ρ is the signed distance from the origin to the line. A pair (ρ, θ) uniquely determines a line, and both parameters are bounded: $|\rho|$ never exceeds the image diagonal, and θ needs only half a turn.

The normal form brings a beautiful **duality**: substitute an image-space point (x_0, y_0) into the equation above and let θ range over $[0^\circ, 180^\circ)$, and the resulting $\rho(\theta) = x_0 \cos\theta + y_0 \sin\theta$ is **a sinusoidal curve** in parameter space — it enumerates all the lines passing through that point. Conversely, if a set of points is collinear on (ρ^*, θ^*), their sinusoidal curves must **all pass through the very same point** (ρ^*, θ^*) of parameter space. Detecting lines thus becomes finding the common intersections of a family of sinusoids.

Why not the more familiar slope-intercept form $y = kx + b$? A vertical line has slope $k \to \infty$, so the **parameter space** is unbounded in the k direction and cannot be discretized into a finite grid; moreover, a uniform partition of k corresponds to wildly non-uniform line directions (k from 0 to 1 sweeps 45°, while from 10 to 11 it sweeps only about 0.5°). The normal form keeps both parameters bounded and is uniform in direction — a parameterization tailor-made for voting.

The engineering realization is the **accumulator**: discretize the (θ, ρ) plane into a grid (e.g. 1°×1 px), initialized to zero; for each edge point, sweep over θ, compute the corresponding ρ, and add one vote to the cell it lands in; once all points have voted, the **peaks** in the accumulator are the lines in the image — the vote count is the number of edge points lying on that line. A global geometric detection problem has been reduced to an array peak search.

Where do the edge points come from? Any edge detector will do (the systematic treatment is Chapter 13) — this chapter thresholds the Sobel gradient magnitude of Figure 15.1 (threshold 200, about 7 of the =8 noise level, anchoring the gate above the noise in the spirit of Chapter 7, with almost no false alarms), yielding **6677 edge points** (Figure 15.2). Note that the gaps in the dashed line and the missing arc of the circle are preserved verbatim in the edge map — the voting is about to face them head-on.

15.2 Reading the Accumulator

Before calling the SDK, we build a 180×1201 accumulator by hand (1°/cell along θ, 1 px/cell along ρ, $|\rho| \le 600$) and let the 6677 points vote one by one. The result is Figure 15.3: θ on the horizontal axis, ρ on the vertical, gray levels stretched by a

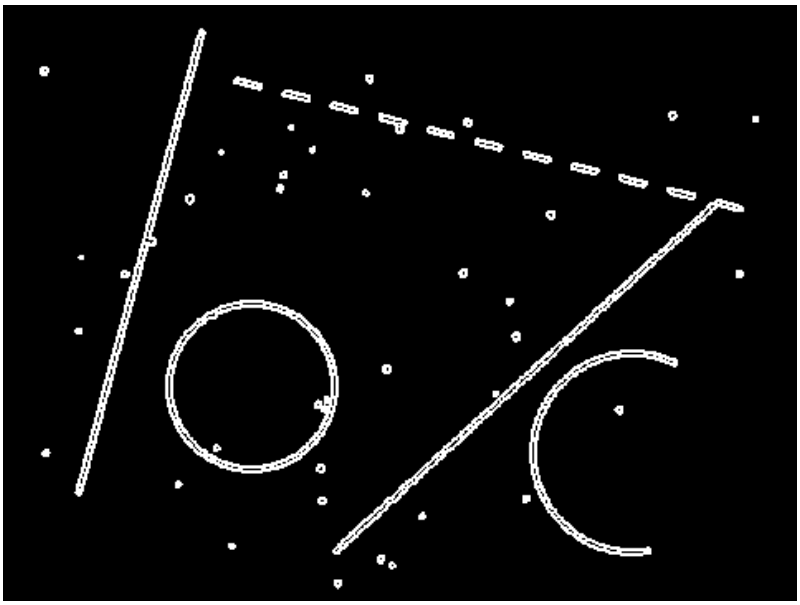


Figure 15.2: Edge map from thresholding the Sobel magnitude (threshold 200 7), 6677 edge points in total. The dashed-line gaps, the missing circular arc, and the stray blobs are all faithfully preserved.

square root. Two layers of structure are visible: the **family of sinusoidal curves** filling the lower half — every faint bright curve is the voting trajectory of one (cluster of) edge points — and, within the family, **three distinctly brighter intersections** — the peaks of the three lines. The votes of the 40 stray blobs scatter along their own sinusoids, share no common intersection, and pile up into a peak nowhere.

Do the peaks land where they should? From the ground-truth endpoints of the three segments we can **predict** each line's (ρ, θ); we then extract the three largest peaks from the accumulator (global maximum + neighborhood suppression, refined to sub-cell accuracy with a 5×5 weighted centroid) and set them against the detections of the SDK's `FindHoughLines`:

Table 15.1: The three ground-truth lines: predicted peak position vs. measured peak of the hand-built accumulator vs. SDK detection

Line	Predicted (ρ, θ)	Accumulator peak $(\rho, \theta, \text{votes})$	SDK detection (ρ, θ)
L1 (solid)	(119.79, 15.00°)	(120.77, 15.30°, 291 votes)	(121.04, 15.01°)
L2 (dashed, 47% gaps)	(9.15, 104.25°)	(9.11, 104.25°, 190 votes)	(9.22, 104.01°)
L3 (solid)	(378.55, 47.60°)	(377.31, 48.05°, 260 votes)	(379.19, 48.02°)

All three lines pass verification: the θ deviation stays within half a degree, the ρ deviation within 1.3 px. That 1.3 px is no mystical error floor; it has a definite origin: the strokes in the scene are about 2.8 px wide, and Sobel detects a row of edge points on **each side** of the stroke; the two rows are each collinear, about 2.8 px apart, and the post-voting peak is a compromise between the two rows of votes, so its position drifts between the sides — the accumulator's peak is forever

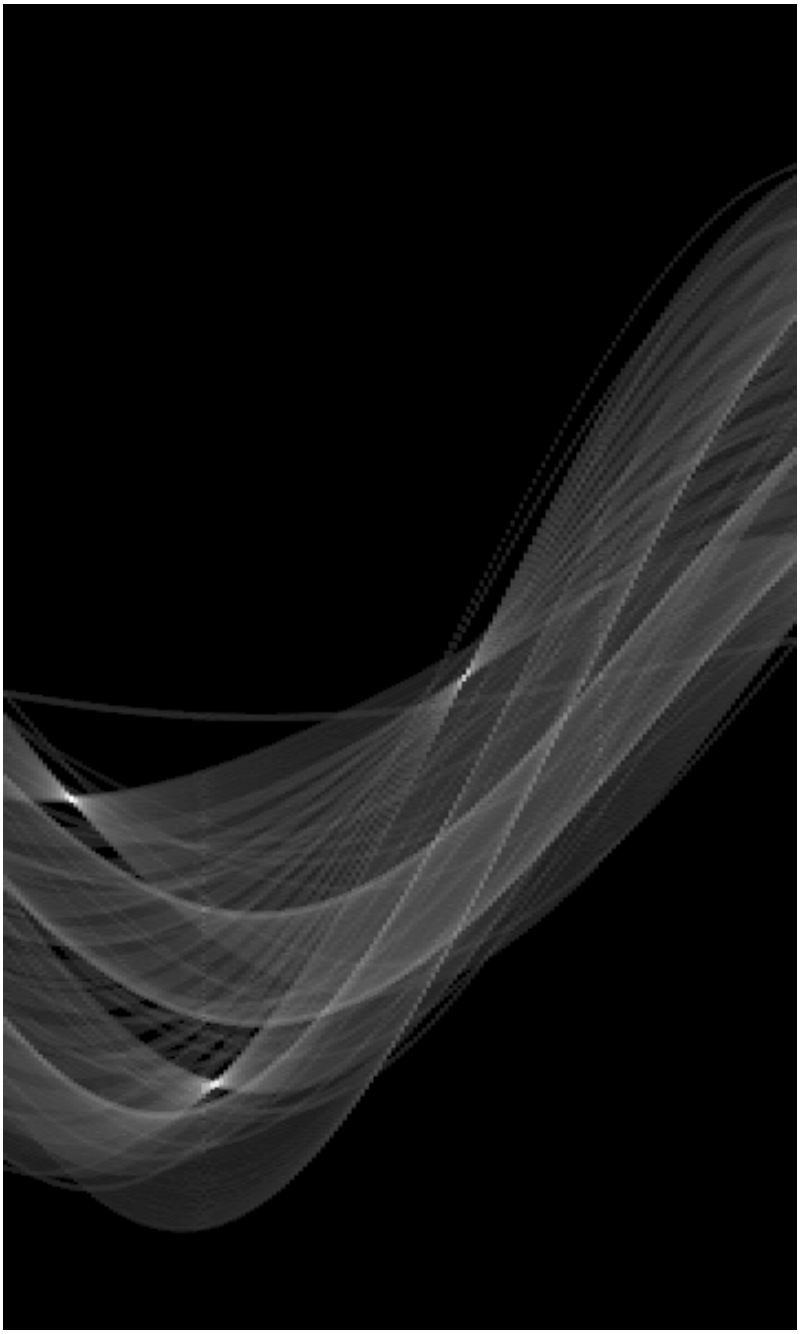


Figure 15.3: The hand-built accumulator (180×1201, horizontal, vertical, square-root gray stretch). Every edge point contributes one sinusoidal voting trajectory; the trajectories of collinear points intersect at a single point, forming three bright peaks corresponding to L1, L2, and L3.

faithful to the geometry of the **edge points**, and the geometry of the edge points is dictated by the stroke width.

The row most worth staring at is L2: the dashed line is missing 47% of its stroke, and its vote count drops from the solid lines' roughly 290 to 190, yet **the position of the peak does not budge and remains clearly identifiable**. This is the soul of the Hough transform — **break immunity**. Voting only asks "does this point lie on this line", never "is it connected to its neighbors": gaps merely shave votes off in proportion; they do not move the peak, let alone make it vanish. By contrast, any method that relies on **local continuity** (edge tracking, contour linking) snaps at the first 14 px gap. Global evidence accumulation and local continuity tracking are two worldviews; when the target is inherently broken, only the former is left standing.

15.3 Multiple Instances and Robustness

What happens if, instead of voting, we hand "find the lines" to our most familiar tool, least squares? Run a negative experiment: fit **one** total-least-squares line to all 6677 edge points at once, and the result is $\rho = 180.04$, $\theta = 93.76°$, with an all-point RMS distance of **82.6 px** — the white line slashing across the image in Figure 15.4, matching none of L1, L2, or L3. This is not an implementation slip; the question itself was wrong: least squares assumes **all points obey one and the same model**, while the points here belong to three lines, two circles, and 40 stray blobs; "the optimal single line over all points" is a compromise for an object that does not exist — it can only land near the centroid of all the structures, oriented along the principal axis of greatest scatter — an "average line" that is nobody.

The Hough transform is born without this dilemma: **each instance gathers its own peak in the accumulator**, none interfering with the others — L1's 291 votes, L2's 190, and L3's 260 each sit in their own place; points that belong to no line — the circle arcs and the clutter — scatter their votes everywhere and never form a peak. Multi-instance separation and

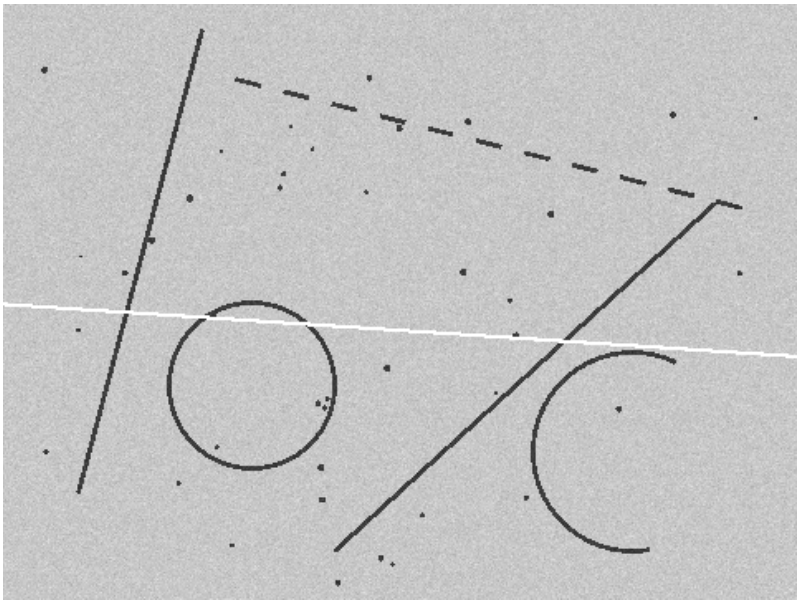


Figure 15.4: The negative experiment: a single-line total-least-squares fit to all 6677 edge points yields =180.04, =93.76°, all-point RMS=82.6 px (the white line) — on multi-instance data the "optimum over everything" is a meaningless compromise.

robustness to clutter are two faces of the **same mechanism**: concentrated votes win, scattered votes lose. Figure 15.5 shows the detections of the SDK's `FindHoughLines`: all three lines are hit (white lines), the dashed one included.

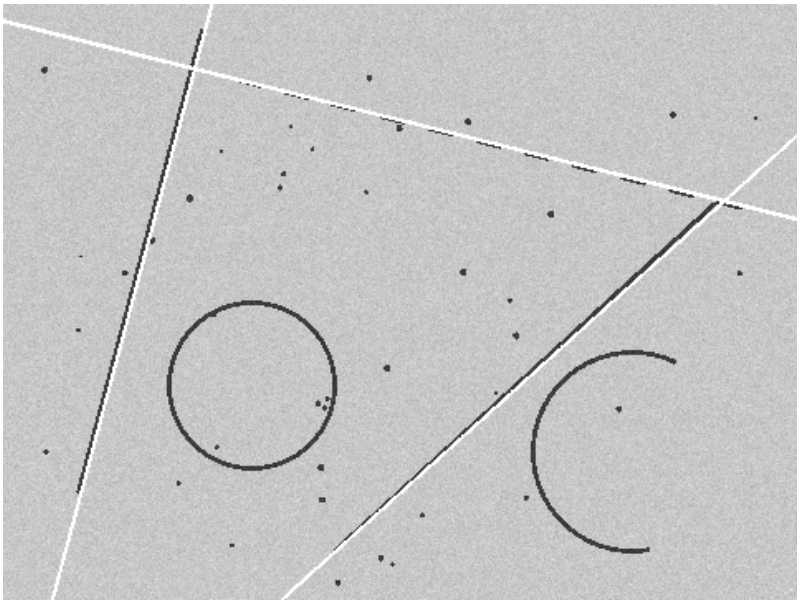


Figure 15.5: The 3 lines detected by the SDK's `FindHoughLines` (white lines; the infinite lines are clipped to the image borders). The dashed line L2 is detected as usual despite its 47% gaps; the circles and the clutter produce no false alarms.

15.4 Hough Circles

A circle has three parameters (c_x, c_y, r), so the accumulator rises from two dimensions to three: an edge point's voting trajectory is no longer a sinusoid but a cone surface in parameter space (the farther a candidate center lies from the point, the larger the required radius). A direct implementation costs an order of magnitude more than the line case in both memory and vote counting, so practical algorithms almost universally accel-

How do Hough and the robust fitting of Chapter 14 (Huber, RANSAC) divide the work? Robust fitting handles "**one** instance + a few outliers", still under the premise that an ROI has roughly framed the target, with subpixel accuracy as the reward; RANSAC can handle multiple instances by "find one, remove its inliers, find the next", but turns inefficient once instances multiply. Hough handles "an **unknown number** of instances + plenty of irrelevant structure", needs no positional prior whatsoever, at the cost of accuracy capped by the accumulator quantization. Rule of thumb: have an ROI and need accuracy — search lines + robust fitting; no prior and need to find them all — Hough.

erate with the **gradient direction**: an edge point's gradient points toward (or away from) the circle center, so votes need only be cast for candidate centers along the gradient ray — the 3D problem collapses into two steps, "2D center accumulation + 1D radius histogram", which is exactly the 2-1 Hough strategy adopted by mainstream implementations (including OpenCV and most industrial libraries).

The experiment continues on the same scene (Figure 15.6): `FindHoughCircles` detects exactly two circles. The complete circle C1 has ground truth $(150, 230, 50)$ and is detected at $(151.0, 231.0, 50.2)$ — a center error of 1.41 px and a radius error of 0.20 px; C2, **missing 40% of its arc**, has ground truth $(380, 270, 60)$ and is detected at $(377.0, 267.0, 57.8)$ — a center error of 4.24 px and a radius error of 2.20 px. The two sets of numbers add up to one sentence: **under occlusion, accuracy degrades, but detection does not fail**. The cause of the degradation is the same as the line's shift, only stronger: the circle's three parameters are mutually coupled, the missing arc tilts the vote mass toward the surviving arc, and the peak gets pulled slightly off; but the remaining 60% of the arc still contributes enough votes to gather into a peak — break immunity replayed on circles.

A feel for the magnitudes: for a 480×360 image and a radius search range of 40 px, a naive 3D accumulator has 480×360×40 6.9 million cells, and every edge point must draw a full circle of votes in every radius layer; with gradient-direction acceleration, each point casts only a few dozen votes along a single ray. Without this step, Hough circles cannot run within a production-line cycle.

The essential ceiling on accuracy must also be seen clearly: measurements show this SDK's circle centers are **quantized to a 2 px grid** and its radii step by about 0.2 px — the direct imprint of the accumulator cell size; no number of votes can buy resolution below the cell. Hence the standard industrial recipe is two-stage: **Hough for coarse localization, search lines for precise measurement** — first use Hough to find every circle across the whole image (no prior needed, occlusion-tolerant), then take the detected center and radius as the prior to construct an annular ROI and hand it to the radial search lines + robust fitting of Chapter 14 for subpixel-precise measurement. Hough answers "where", search lines answer "how precisely" — the methods of the two chapters are not rivals but upstream and downstream of one pipeline.

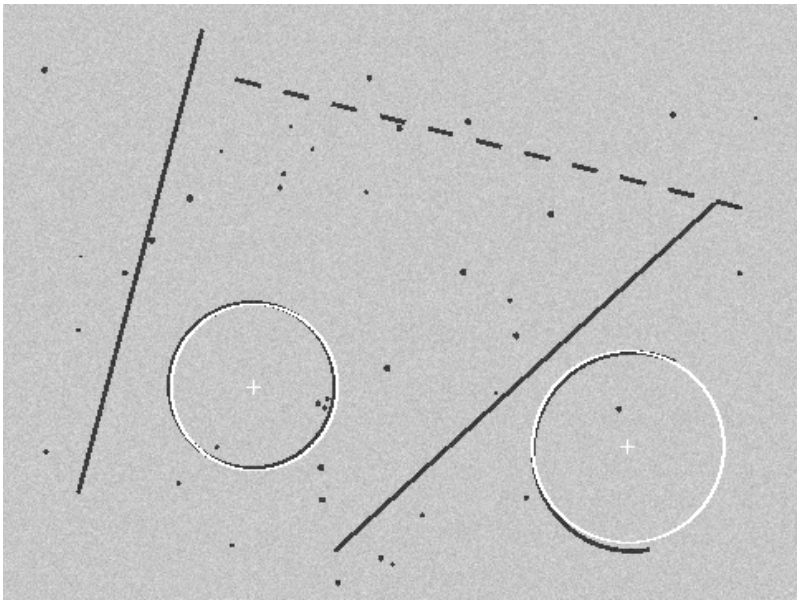


Figure 15.6: Detections of the SDK's `FindHoughCircles` (white circles and crosses mark the detected circles and centers). The complete circle C1 has errors of 1.41/0.20 px; C2, missing 40% of its arc, has errors of 4.24/2.20 px — under occlusion accuracy degrades but detection does not fail.

15.5 SciVision Implementation

Hough lines are provided by `SCIMV::SciSvHoughLines`, which performs Canny edge extraction internally and consumes the gray image directly:

```
SCIMV::SciSvHoughLines houghL;
SciPointArray term;
SciVarArray lineAng, lineLen;
long rc = houghL.FindHoughLines(src, fullROI, /*cannyLow*/ 60, /*cannyHigh*/ 120,
    /*accumThreshold*/ 65, /*minAngle*/ -90, /*maxAngle*/ 90,
    /*AngleGap*/ 5, /*DistGap*/ 10, /*maxNum*/ 6, /*AngleCheck within range*/ false,
    &term, &lineAng, &lineLen);
```

`cannyLow/cannyHigh` are the dual thresholds of the internal Canny; `accumThreshold=65` is the vote-count gate — peaks with fewer than 65 votes are not reported; `minAngle/maxAngle` bound the accepted range of line angles; `AngleGap=5` and `DistGap=10` define the neighborhood for peak deduplication (peaks differing by $<5°$ in angle and <10 px in distance are merged into one line); `maxNum=6` caps the number of lines returned. In this example up to 6 lines were allowed and exactly the 3 ground-truth lines were detected — no false alarms under these parameters.

In actual testing this API exhibits three behaviors that contradict intuition (or the documentation) and must be recorded honestly. **First, the header file does not state what the values of `AngleCheck` mean**: measurement shows that `false` means "find lines **within** the angle range" and `true` means find those **outside** it — passing `true` together with the full range [-90,90] directly yields the "no suitable line found" error code 122408001. **Second, the default Canny thresholds 20/40 fail on this scene**: under $\sigma=8$ noise there are too many false edge points, the votes of the real lines drown in noise votes, and phantom diagonal lines spanning the whole image surface in the accumulator; results only stabilize once the thresholds are raised to 60/120 — the Canny gate must be anchored above the noise level, the same reasoning as the 7σ threshold of the hand-built edge map in Section 15.1. **Third, the semantics**

of the return values: the "endpoints" in `term` are not segment endpoints but **the intersections of the infinite line with the image borders** (which is why every white line in Figure 15.5 runs clear across the image), and `lineLenth` is the clipped length; the angle returned in `lineAng` is $90^\circ - \theta$ (the line's direction angle, not the normal angle) — do not get the conversion backwards.

Hough circles are provided by `SCIMV::SciSvHoughCircles`:

```
SCIMV::SciSvHoughCircles houghC;
SciPointArray centers;
SciVarArray radii;
rc = houghC.FindHoughCircles(src, fullROI, /*minDist*/ 100, /*edgeThreshold*/ 160,
    /*accumThreshold*/ 65, /*minRadius*/ 35, /*maxRadius*/ 75, /*maxNum*/ 4,
    &centers, &radii);
```

`minDist=100` is the minimum spacing between two circle centers (deduplication); `edgeThreshold` is the strength gate of the internal edge extraction; `accumThreshold=65` is the vote-count gate of the center accumulator; `minRadius/maxRadius` delimit the radius search band of 35–75 px. There is a pitfall here as well: **the default `edgeThreshold=80` is too low** — noise edges gather into **phantom circles whose radius hugs `maxRadius`** (a large-radius circle has a long circumference, more noise points happen to pass by, a built-in vote advantage); after raising it to 160 and pairing it with `accumThreshold=65`, only the two ground-truth circles are detected even with `maxNum=4` left open. A final reminder about the output quantization granularity: centers land on a 2 px grid and radii step by about 0.2 px — the moment you receive the result you should know it is a "coarse localization", and leave the precise measurement to the downstream. The complete project that generates all of this chapter's images and numbers lives in `code/hough_transform/`.

Industry Case: Counting Pins on a Tray

An IC tray inspection station must count the pins of each chip in its pocket: under ring light the metal pins reflect and image as a set of **intermittent short bright lines** — each pin breaks into two or three segments, and the count varies with

the chip model. Template matching does not apply (the pin count is variable and the appearance changes with the reflections), and search lines have nowhere to lay their ROIs. Hough lines fit exactly: the pin direction is known (perpendicular to the chip body), so narrowing `minAngle/maxAngle` to an angular band of $\pm 10°$ around that direction throws out reflections in irrelevant directions from the start; the several broken bright segments **pool their votes into the same peak** in the accumulator — break immunity automatically merges "several segments" back into "one pin"; finally the vote-count gate filters out sporadic reflections, and the number of detected peaks is the pin count. The center of gravity in tuning is exactly the same as in this chapter's experiments: first measure the noise level of the production-line images and anchor the internal Canny thresholds above it — the moment phantom lines appear, the count can no longer be trusted.

15.6 Summary

- **The Hough transform turns geometric detection into voting**: under the normal form $\rho = x\cos\theta + y\sin\theta$, an image point is dual to a sinusoid in parameter space, and the sinusoids of collinear points intersect at a single point; a discrete accumulator counts the votes, and the peaks are the lines.
- **Break immunity is the Hough transform's soul**: voting only checks "on the line or not", never "connected or not" — gaps cut votes but do not move peaks. The dashed line with 47% gaps still peaked with 190 votes, and the 40%-occluded circle was detected as usual.
- **Multi-instance separation and clutter robustness are the same mechanism**: each instance gathers its own peak, scattered votes gather into none; in the negative experiment, a single-line least-squares fit to the 6677 points produced an "average line" with RMS 82.6 px — on multi-instance data the optimum over everything is a meaningless compromise.
- **Hough accuracy is capped by the accumulator quantization** (here circle centers on a 2 px grid and

radii in 0.2 px steps; the line's deviation is further swayed by the votes from the two sides of the stroke), so the engineering recipe is "Hough for coarse localization + search lines for precise measurement", forming an upstream–downstream pair with Chapter 14.

- **Anchor parameters to the noise and verify semantics empirically**: internal Canny thresholds below the noise level breed phantom lines/circles; this chapter's SDK has `AngleCheck` semantics opposite to the documentation, returns border-clipped intersections as endpoints, and reports the angle as $90^\circ - \theta$ — always validate critical semantics on synthetic images with known ground truth before going live.

The original reference for the Hough transform is the classic paper by Duda and Hart (Duda and Hart 1972); its generalization from lines to arbitrary shapes is Ballard's generalized Hough transform (Ballard 1981), and Illingworth and Kittler's survey systematically maps the variants and computational strategies (Illingworth and Kittler 1988). For its connection to geometric measurement pipelines, see further the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

16 Template Matching

"Where is this thing in the image?" — that is the question industrial vision gets asked most often: where on the tray is the part the robot arm must pick? By how much is the pad the placement head must align to offset? Where does the measurement program's fiducial land in this frame? The most direct answer to this question is to take a "photo" of the target and compare it against the image position by position, looking for the spot that resembles it most. This is **template matching**: take a small reference image as the **template**, slide it across the search image comparing as you go, and output the position (and angle) of highest similarity. It needs no modeling, no training samples, and works on arbitrary texture — it is, deservedly, the workhorse of industrial localization.

This chapter runs from start to finish on a set of real sample images: surface-mount aluminum electrolytic capacitors (rated 470 µF, 35 V, RVT series) scattered on a black background, provided by the Smart3 gray-value-matching example recipe (1280 × 960 single-channel grayscale). Each capacitor is a reflective metal disc whose top face carries the silkscreen "470 / 35V / RVT" in three lines, with a dark polarity stripe on the left — stripe and text together break the circular rotational symmetry, giving the angle a unique solution. The template is cropped from one capacitor in the upper left of sample 001 (Figure 16.1a): a 230 × 230 window around its disc top (disc diameter about 214 px). The search image is sample 003 (Figure 16.1b), on which five capacitors of the same type are scattered at **differing positions and poses**: some upright, some rotated by small angles, and one flipped by nearly 180° — exactly how parts look after being fed onto a line. Section 16.5 further validates the same model on samples 004 and 005.

(a) Template: one capacitor disc cropped from sample 001

(b) Search image (sample 003): five same-type capacitors at differing poses

Figure 16.1: The experimental data for this chapter. (a) The 230×230 template; the dark polarity stripe and the silkscreen text leave it with no rotational symmetry; (b) the 1280×960 search image: five 470 µF / 35 V capacitors scattered on black, each at a different position and rotation, including one flipped by nearly 180°.

16.1 Similarity Measures

Align the top-left corner of the template $t(u, v)$ with position (x, y) of the search image f; "how alike is it?" needs a number for an answer — this is a **similarity measure**. The two plainest are accumulations of pixel-wise differences: the **sum of absolute differences (SAD)** and the **sum of squared differences (SSD)**:

$$D_{\mathrm{SAD}}(x,y) = \sum_{u,v} \big| f(x{+}u,\, y{+}v) {-} t(u,v) \big|, \qquad D_{\mathrm{SSD}}(x,y) = \sum_{u,v} \big(f(x{+}u,\, y{+}v) {-} t(u,v) \big)^2 .$$

For both, smaller means more alike; they are cheap to compute and plain in meaning, but they carry one defect that is fatal on a production line: **sensitivity to brightness changes**. Suppose the window content somewhere happens to be the template multiplied by an illumination gain g (lights getting stronger or weaker, batch-to-batch differences in workpiece surface reflectance — all produce this kind of linear change); then

$$D_{\mathrm{SSD}} = \sum_{u,v} \big(g\, t(u,v) - t(u,v) \big)^2 = (g-1)^2 \sum_{u,v} t(u,v)^2 .$$

The shape matches to the last detail, yet the residual grows **quadratically** with the gain deviation — let the lighting shift, and a "perfect match" looks "far off" in the eyes of SSD.

The remedy is to "standardize" the window and the template separately before comparing: subtract the mean (removing the brightness offset), divide by the standard deviation (removing the gain) — what remains is pure "shape." This is **normalized cross-correlation (NCC)**:

$$\rho(x,y) = \frac{\displaystyle\sum_{u,v} \big(f(x{+}u,\, y{+}v) - \bar{f}_{x,y} \big) \big(t(u,v) - \bar{t}\, \big)}{\displaystyle\sqrt{\sum_{u,v} \big(f(x{+}u,\, y{+}v) - \bar{f}_{x,y} \big)^2} \, \sqrt{\sum_{u,v} \big(t(u,v) - \bar{t}\, \big)^2}},$$

where $\bar{f}_{x,y}$ is the mean of the current window and $\bar{t}$ is the template mean. Its invariance to linear illumination changes can be seen through in one line: replace the window content by $af + b$ (with $a > 0$); the mean subtraction eliminates the offset b in one step, and the gain a, appearing as a common factor in numerator and denominator, cancels — ρ does not budge.

The experiment confirms the derivation. Scanning sample 003 pixel by pixel with the **fixed** 0° **template** cropped from sample 001, the highest response lands on the nearly upright instance in the upper left, with an NCC of 0.827. That this does not reach close to 1 is precisely because the instance itself still carries about 12° of residual rotation relative to the template (see the angle-search result in Section 16.5) — a fixed-angle template cannot even align the "best" instance, a foreshadowing we defer to Section 16.3. To isolate NCC's illumination invariance, we apply a **controlled and openly disclosed** perturbation to the real image: multiply the gray values within this instance's neighbourhood wholesale by 0.5, emulating the illumination at this station dropping by half. The result is plain — NCC moves from 0.827 to 0.827 (a change of 0.0001), not budging; yet the root-mean-square SSD residual (rms) at the same position rises from 41.2 to 62.3, about a factor of 1.5. If a detection threshold with 40% margin is calibrated on the bright instance, the verdicts come out as follows:

A geometric view: regard the mean-subtracted window and template each as a high-dimensional vector; NCC is then the **cosine** of the angle between the two vectors, hence always $|\rho| \le 1$; $\rho = 1$ holds if and only if the window is a positive linear transform of the template, and $\rho = -1$ corresponds to a black-white inversion (opposite polarity) — exactly the situation the SDK's `polarity` parameter distinguishes.

Table 16.1: Detection verdicts of SSD versus NCC under an illumination change, at the same threshold

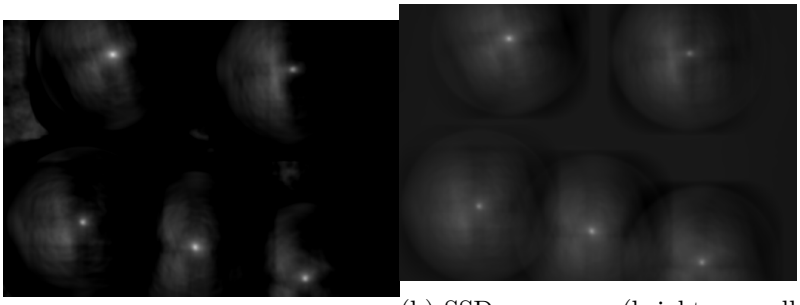
Measure and threshold	Bright instance	50% illumination instance
SSD (rms ≤ 58)	detected (41.2)	**missed** (62.3)
NCC ($\rho \ge 0.80$)	detected (0.827)	detected (0.827)

One utterly ordinary decay of the lighting is enough for SSD to declare a good part "absent." NCC carries a further engineering benefit: its score naturally lies in $[-1, 1]$ (the SDK converts it to a 0–100 scale), so a "pass score" threshold transfers directly across stations and cameras; an SSD threshold, by contrast, depends on the gray-value scale and the noise level — swap the camera and you must recalibrate. This is why industrial

matching almost universally uses NCC (or the shape matching of the next chapter).

16.2 Score Maps and Peaks

Arrange the similarity at every position into an image and you obtain the **score map**. Figure 16.2 shows the NCC score map and the SSD score map of the same scene (the latter displayed as "bright = small residual"). To bring the cost of a full-resolution full-image scan down to a demonstrable range, both maps are computed at 1/2 resolution (640×480) with a 115×115 template (the reason is given in Section 16.4).



(a) NCC score map

(b) SSD score map (bright = small residual)

Figure 16.2: Score maps from a full-image scan with the fixed 0° template. (a) NCC: each of the five capacitors stirs up a response, but **the peak height falls off with rotation** — the upright instance at 0.827 is the brightest, sharpest peak, and the more an instance is turned, the lower its score; (b) SSD: again the upright instance is the brightest, while the flipped instance is far dimmer.

On the NCC score map each of the five capacitors raises a circular response, but **the peaks are clearly graded**: under the fixed 0° template the five instances' peak values descend from 0.827 (upright) to 0.750, 0.656, 0.622, and 0.516 — the more an instance departs from the template pose, the lower the correlation, and the instance flipped by nearly 180° is left with only a faint response just over 0.5. This is precisely the lead-in

to the next section: NCC is helpless against rotation. The SSD map tells the same story in its "residual" version: the upright instance is the brightest blob, and the brightness of the rest collapses fast.

The score map also reveals a deeper fact: **the sharpness of a peak is the localization power**. The sharper the peak, the faster the score falls off around the optimum, and the less room noise has to nudge the summit aside; conversely, a gentle ridge-like response means a string of positions are "about equally alike," and localization is bound to wander. The richer the high-frequency structure in the template (disc edge, polarity stripe, silkscreen text), the sharper the peak. Finally, the integer-pixel summit is only the starting point: within the peak's neighborhood, apply three-point parabola vertex interpolation to the scores along x and y separately (the $\delta = (g_{-1} - g_{+1})/(2(g_{-1} - 2g_0 + g_{+1}))$ derived in Chapter 2) to refine the position to subpixel — fractional coordinates such as the $(404.07, 204.06)$ reported by commercial matchers come exactly from this.

16.3 Rotation and Pose Search

NCC solves illumination but is helpless against rotation: on sample 003 the fixed 0° template scores **only 0.827 even on the best-aligned instance** (which itself is still turned by about 12°), and the instance flipped by nearly 180° is left with barely over 0.5 — at a threshold of 0.80, the very same capacitor is declared absent merely for having turned. NCC compares with pixel-wise alignment; turn the template and the edges misregister, and the correlation collapses rapidly. A practical matcher must therefore make angle (and, when necessary, scale) an explicit search dimension: prepare a rotated copy of the template for each candidate angle, compare them one by one, and take the pose with the highest score.

SciVision's approach moves this expense to **training time**: `CreateGreyImageModel` pre-rotates the template into a series of sampled poses according to `startAngle`, `angleExtent`, and `angleStep` and stores them in the model. Since capacitors may

land at any orientation after feeding (the sample even contains an instance flipped by nearly 180°), this example uses a **full-circle search**: starting at −180°, spanning 360°, with a step of 1°, for 360 poses in all; at search time each pose is scored, and the angle is then refined by interpolation around the best pose. The cost is proportional to the number of angles — the wider the range and the finer the step, the slower the training, the larger the model, and the more time each match takes. The immediate payoff: the upright instance that scored only 0.827 under the fixed template climbs to 97.58 (0–100 scale) once angle search is on — align the angle and the correlation fills right back up.

How to choose the step? Too coarse, and when the true angle falls between two sampled bins the score drops and the localization drifts; too fine is sheer waste. A practical estimate: a rotation by $\Delta\theta$ displaces a point at radius r by about $r\,\Delta\theta$ (in radians); requiring the displacement at the template's outermost edge to stay within 1 px gives $\Delta\theta \approx 57.3^\circ/r$ — this example's disc has a radius of about 107 px, so a 1° step corresponds to an outer-edge displacement of about 1.9 px, already a touch coarser than the "1 px rule," with the subpixel angle interpolation at search time making up the rest.

16.4 Pyramid Acceleration

The cost of a full-resolution pixel-by-pixel scan is staggering: in this example, $(1280 - 230 + 1) \times (960 - 230 + 1)$ windows at about 5.3×10^4 multiply-adds each, totaling about 4×10^{13} operations — unbearable even for a single angle, which is exactly why pyramids and commercial matchers exist (and why the score maps of the previous section retreat to 1/2 resolution for computation). The way out is the **image pyramid** and the **coarse-to-fine** strategy: repeatedly downsample the scene and the template by 2×2 averaging (box pre-filter, then decimate — precisely the anti-aliasing downsampling emphasized in Chapter 10), run the full-image scan on the smallest top level — nearly free; take a handful of coarse candidates, map them back up level by level, and refine only within small windows around the candidates.

In the experiment we use 1/2 resolution (640×480, template 115×115) as the baseline for comparison: a full-image NCC scan on the baseline is about 2.5×10^9 operations, measured at about 10.6 s (an unoptimized /Od build, so the absolute value runs high and fluctuates with machine load, but the ratios are representative). On top of this baseline we hand-write a two-level pyramid: the top level downsamples scene and template two more levels (top 160×120, template 28×28) for a full scan, keeps 3 candidates, and each returns to the baseline resolution for refinement within a ± 8 px window. The result: about 0.11 s against the full scan's 10.6 s — **about a 100-fold speedup** — while the peak positions found agree with the baseline full-resolution scan **pixel for pixel**: not a cent of accuracy lost.

The cost accounting: with each level down, the scene pixel count and the template area each shrink to 1/4, so the per-level scan cost shrinks to 1/16; after descending two levels, the top-level full scan is only 1/256 of the original workload, and the candidate-refinement windows (here ± 8 px) cost a drop in the bucket. Every added level pays off by an order of magnitude.

Can the levels be added without limit? No. The constraint is that **the template at the top level must remain recognizable**: when the 230×230 capacitor disc shrinks to a few tens of pixels the circular outline and the polarity stripe still survive; shrink further and the stripe and silkscreen smear away completely — the candidates the top level produces are then untrustworthy. The SDK's `GetGreyAutoPyramidLevel` makes this trade-off for you — for this template it automatically suggests 5 levels (counting the original-image level), with the top-level template at about 14×14 where the disc and stripe are still discernible.

16.5 SciVision Implementation

Gray-value matching in SciVision is carried by `SCIMV::SciSvGreyMatch`, in two steps — training and search:

```
SCIMV::SciSvGreyMatch gm;
int level = -1;
gm.GetGreyAutoPyramidLevel(tmplImg, mask, &level);   // automatically suggests 5 levels for thi
SciMatchModel model;
gm.CreateGreyImageModel(tmplImg, mask, /*pyramidLevel: automatic*/ -1,
                        /*startAngle*/ -180, /*angleExtent*/ 360, /*angleStep*/ 1, &model);

SciPointArray centers; SciVarArray angles, scores;
```

```
gm.FindGreyImageModel(sceneImg, searchROI, model,
                      /*minScore*/ 60, /*matchCount*/ 5, /*overLapRatio*/ 50,
                      /*startAngle*/ -180, /*angleExtent*/ 360,
                      /*interpMethod: bilinear*/ 1, /*polarity: distinguish polarity*/ 0,
                      /*endLevel: refine down to the original level*/ 0, /*clutter*/ 0.0f,
                      &centers, &angles, &scores);
```

On the training side: passing -1 for `pyramidLevel` lets the SDK choose the level count automatically (i.e., the result of `GetGreyAutoPyramidLevel`); `startAngle`/`angleExtent`/`angleStep` are the three angle-sampling parameters discussed in Section 16.3, here covering the full circle. On the search side: `minScore` is the acceptance threshold on the 0–100 scale (candidates below 60 are discarded outright); `matchCount` caps the number of instances returned (5 here, for the five capacitors); `overLapRatio` controls how much overlap two results may share, for multi-instance deduplication; the angle interval matches the training one; `interpMethod=1` rotates the template with bilinear interpolation; `polarity=0` requires the light-dark polarity to agree (a black-white-inverted "part" does not count as a match); `endLevel=0` means refine all the way down to the original-image level — if the localization accuracy requirement is loose, stopping at a higher level saves further time; `clutter` penalizes cluttered background outside the template region, with 0 meaning off.

Two pitfalls hit in actual testing deserve an honest record. **First, the ROI semantics of `mask`.** The matching family is not uniform in what it demands of the `mask` parameter: for feature matching, color matching, and shape matching (`SciSvFeatureMatch`/`SciSvColorMatch`/`SciSvScaleShapeMatch`), `Create*Model` **must be passed a default-constructed UNDEF ROI** — passing a rectangle covering the whole image instead triggers errors such as "insufficient feature points"; gray-value matching, by measurement, accepts a `GenRect1` rectangle covering the entire template, but be sure to remember that `GenRect1`'s bottom-right corner is an **exclusive** endpoint — for the full template, pass (T, T), not $(T-1, T-1)$. When switching to another matcher in the same family, first confirm which kind of mask it expects. **Second,**

the angle convention. The SDK returns angles in the **mathematically positive direction (counterclockwise positive)**: the upright instance in sample 003 is reported as $+12.00^\circ$, and feeding it straight into rotation drawing on the y-down image side turns things the wrong way; negate to convert. The correctly oriented pose boxes in this chapter's `matches.png` (Figure 16.3) were drawn precisely after that negation.

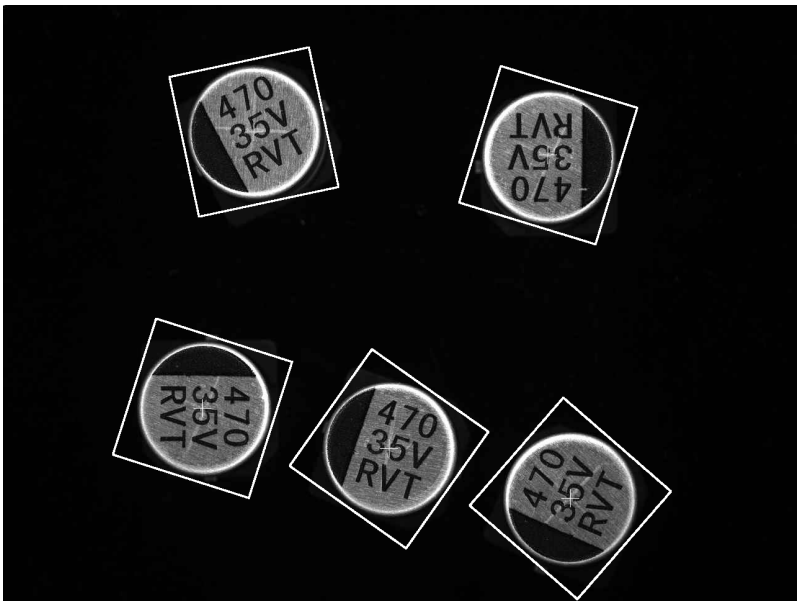


Figure 16.3: The search results of `FindGreyImageModel` on sample 003: all five capacitors found — crosses mark the subpixel centers, white boxes mark the matched poses — including the nearly 180° flipped instance and the large-angle rotated ones; none is falsely detected on the black background.

The SDK's measured results on sample 003 are as follows (angles in the SDK's counterclockwise-positive direction):

Table 16.2: Measured results of SDK gray-value matching (sample 003, minScore=60, angleStep=1°)

Instance	Position (x, y)	SDK angle	Score
0 (upright)	$(404.07, 204.06)$	$+12.00°$	97.58
1	$(621.79, 714.39)$	$-35.26°$	92.87
2 (flipped)	$(878.39, 241.39)$	$+163.00°$	91.15
3	$(914.14, 794.30)$	$+49.00°$	90.70
4	$(321.08, 649.11)$	$-107.66°$	89.59

All five capacitors score above 89, the angles span the full circle (including a $+163°$ flipped instance), and the positions are refined to subpixel. Two cross-checks are worth noting. First, the highest peak found by the hand-written fixed $0°$ template at 1/2 resolution (upsampled back to $(404, 204)$) agrees **pixel for pixel** with the SDK's instance 0 at $(404.07, 204.06)$ — two independent implementations point to the same position. Second, for that same upright instance the fixed-angle NCC is only 0.827, whereas with angle search on the SDK gives 97.58 — **angle search recovers the score**, exactly the argument of Section 16.3. The same model also finds **all five** capacitors on samples 004 and 005 (lowest scores 88.9 and 91.2 respectively), confirming the model's robustness to position and pose changes. The complete runnable project is at `code/template_matching/`.

Industry Case: Feed Rotation Beyond the Angle Search Range

On an assembly line, a gray-value matching model had its `angleExtent` set to $\pm 20°$ at training time according to the incoming material of the day, and it ran stably for months. Then the feeding method changed and the workpieces' actual rotation reached $\pm 35°$ — instances beyond the search range had no template pose that could line up at all, match scores plunged below the threshold, and entire batches went undetected. The first reaction was to widen the range straight to the full circle "to be safe"; the result was a marked rise in training time and per-match latency, and worse, with the pose count doubled, similar shapes had more chances to collide with some angle, and the false-detection rate crept up instead. The final

solution was to measure the actual rotation distribution of the upstream mechanism, set the range from its envelope plus margin, and add guide rails on the feed track to physically force the rotation back within range. The lesson: the angle search range's job is to **cover physical reality** — in this chapter's samples the capacitors do flip, so the full circle is warranted; but for a pose-constrained station it must not be smaller than the material's true rotation, and it is emphatically not "the bigger, the safer."

16.6 Summary

- Template matching = **similarity measure + pose search**: the measure decides which disturbances it is robust to, and the search dimensions decide which pose variations it can tolerate — neither can be spared.
- **SSD/SAD are sensitive to brightness changes** (a gain g brings a residual term of $(g-1)^2$); **NCC subtracts the mean and divides by the standard deviation, and is strictly invariant to linear illumination** — measured: dropping one capacitor's neighbourhood illumination to 50% moves NCC merely from 0.827 to 0.827, while the SSD residual grows 1.5-fold and causes a miss.
- **The sharpness of the score-map peak is the localization power**; the integer-pixel summit is refined to subpixel by parabola vertex interpolation — in this example the SDK localizes all five capacitors to subpixel with scores above 89.
- **NCC has no rotation invariance** (under a fixed 0° template the upright instance scores only 0.827 and the flipped instance drops to just over 0.5), so angle must be searched explicitly; here the full circle covers the capacitors' arbitrary orientation, and with search on the score climbs from 0.827 to 97.58.
- **Coarse-to-fine pyramids are the standard means of speeding up matching**: a coarse top-level scan plus candidate refinement gave a measured ~100-fold speedup with peak positions exactly identical to the full scan; the ceiling on the level count is that the top-level template

must remain recognizable (5 levels automatically here).

NCC's robustness boundary stops at **linear** illumination: under nonlinear changes such as specular glare and local shadows, and when the part is partially occluded or changes scale, gray-value matching scores all collapse — those scenarios call for edge-based shape matching, see Chapter 17. How this chapter's normalized cross-correlation can be computed efficiently with integral images is covered in Lewis's classic note (Lewis 1995); for template matching within the larger picture of image registration, see Brown's survey (L. G. Brown 1992). For a more systematic treatment of gray-value matching and its acceleration strategies, see further the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

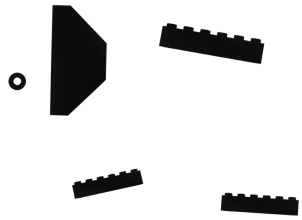
17 Shape Matching

The previous chapter (Chapter 16) crowned NCC as the workhorse of industrial localization, and also exposed its boundaries at the end: NCC resists neither **rotation** nor **occlusion**, and it cannot hold the line against **nonlinear illumination**. On a production line these three regular visitors cross the boundary again and again — incoming parts arrive at random angles, grippers and carrier tapes cover part of the workpiece, and specular highlights and oil films warp the gray-value mapping into an arbitrary curve. All three failures share one root cause: gray-value matching compares **the gray value of every single pixel**, and gray values are precisely the least stable thing in an image — and the pixel patch itself is nailed to one orientation and one size. The way out is to change what we compare — not gray values, but **shape**: abstract the template into a set of edge points (position + gradient direction), search over a pose grid of angle and scale, and at each point check only whether the image's gradient direction agrees. This is **shape-based matching**, the most interference-resistant localization algorithm in commercial vision libraries.

This chapter runs its experiments on a set of real industrial sample images. The contour-matching images come from a localization recipe's silhouette samples (Figure 17.1): black parts on a white background, with the lead role played by a **pentagonal wedge** — a vertical left edge, a near-horizontal top edge, a tip converging to the right, and a chamfered corner at the lower right; the flat edges and the chamfer make it strongly **non-rotationally-symmetric**. The supporting cast includes a ring washer and a toothed rack as distractors. Across the five sample images the wedge is scattered at a wild variety of angles (Section 17.2), exactly testing rotation robustness and clutter rejection. The scale experiment uses a separate set: the

same pile of parts (a screw, a knob, a square connector) photographed at three working distances, the parts shrinking from near to far as the camera pulls back (Section 17.4).

(a) Template: the pentagonal wedge (cropped from sample 1, 195×323)



(b) Search image: sample 1, wedge + ring + toothed-rack distractors

Figure 17.1: The real sample images of this chapter's contour matching. (a) The pentagonal wedge cropped from sample 1 as the template — the vertical edge plus the chamfer make it non-rotationally-symmetric; (b) the 1006×759 search image (original 4024×3036, downsampled by 4), with the wedge sharing the frame with a ring, a toothed rack, and other distractors.

17.1 Similarity Based on Gradient Direction

The model in shape matching is not a pixel patch but a **list of edge points**: at training time, edge extraction is run on

the template (the hysteresis-threshold pipeline of Chapter 13), yielding n edge points p_i, each carrying the **gradient direction** θ_i at that location. At search time the list is transformed onto the image under a candidate pose (translation + rotation + scale); at each point's landing position the image's gradient direction is read out and differenced against the model direction (rotated along with the pose) to give $\Delta\theta_i$, and the score is the average direction agreement:

$$s = \frac{1}{n}\sum_{i=1}^{n} \cos \Delta\theta_i .$$

When the directions agree perfectly, $\cos \Delta\theta_i = 1$ and $s = 1$ (which the SDK reports on a 100-point scale). When directions are random, positive and negative contributions cancel and tend to zero. Hidden inside this one formula are shape matching's three great weapons.

Naturally searchable over rotation and scale, because the model is a list of points with geometric coordinates: rotate it by ϕ and scale it by σ as a whole and you get the prediction for a new pose, so scoring pose by pose localizes across continuous angle and scale — exactly what a gray-value patch cannot do: a patch must be resampled the moment it rotates, and the NCC formula has no degree of freedom for angle in the first place.

Robust to illumination, because gradient direction does not change under gray-value gain and offset: replace the image with $af + b$ $(a > 0)$ and the gradient vector becomes $a\nabla f$ — its length changes, but **its direction does not budge**. Better still, even when the gray-value mapping is nonlinear, as long as it is locally monotonic, the course of the iso-gray contours is unchanged and the direction is still conserved. NCC holds the line at linear; gradient direction holds it even for smooth nonlinearities.

Natural tolerance of occlusion, because the score is an **average** of per-point contributions: model points that land on the occluder see a gradient direction unrelated to the model, so their contributions are random within $[-1, 1]$ with near-zero mean — missing points merely **dilute** the score without disturbing the contributions of the remaining visible points in the

When $a < 0$ (light–dark inversion), the gradient direction flips by 180° and $\cos \Delta\theta_i = -1$ — the score is penalized instead. Whether to score by cos (requiring consistent **polarity**) or by |cos| (ignoring polarity) is controlled by the SDK's `polarity` parameter: specular reflection on curved surfaces and backlight switching flip local polarity, and only then should it be opened up to 2.

slightest. If 70% of the part is visible, roughly 70% of the score remains. Contrast NCC: occluded pixels enter the sum of squares at full weight, carrying enormous gray-value residuals, and a small amount of occlusion is enough to drag the correlation beyond recognition. One is "majority rules," the other is "one vote vetoes" — and that is where the gap in robustness comes from.

17.2 Rotation and Clutter Experiment

Build a model from the wedge in sample 1 with `SciSvScaleShapeMatch` (angle range $\pm 180^\circ$, step 1°, scale fixed at 1.0) and search the full image of all five samples with minScore = 40. Six wedges are scattered across the five images (sample 3 holds two in one frame), and **all six are hit**, with angles spanning from -152° to $+102^\circ$ — a full 254° range of rotation:

Table 17.1: Measured contour-matching results (template from sample 1, minScore=40, subpixel=0)

Sample	Instance	Position (px)	Solved angle	Score
1	Wedge (template source, upright)	$(318, 237)$	0°	99.8
2	Wedge (large rotation)	$(455, 497)$	-152°	99.8
3	Wedge A	$(325, 196)$	102°	99.6
3	Wedge B	$(323, 570)$	89°	93.7
4	Wedge (near-upright)	$(702, 252)$	1°	99.8

Sample	Instance	Position (px)	Solved angle	Score
5	Wedge (rotated)	$(310, 209)$	$-68°$	95.5

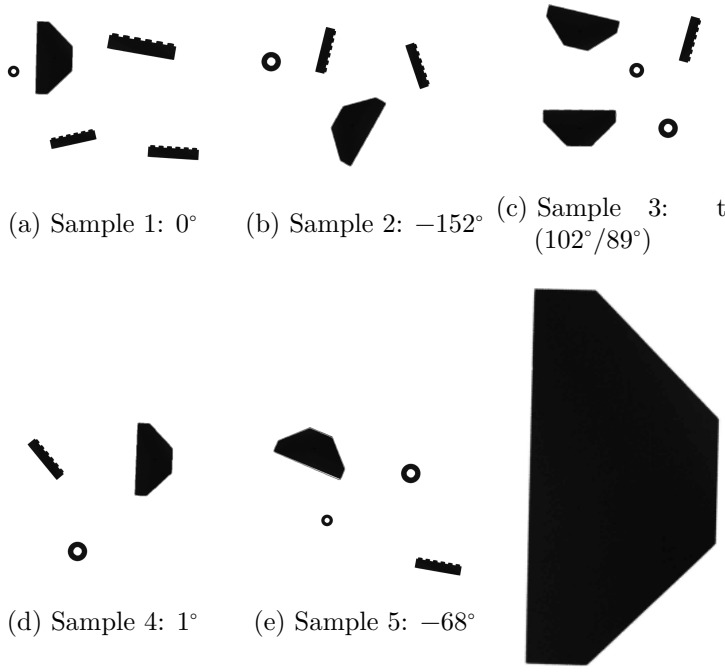
Positions are pixel coordinates in the 1006×759 downsampled image. The SDK returns angles in the mathematically positive direction (counterclockwise positive); they are negated when rendering in image coordinates (y downward). The silhouette edges have extremely high contrast, so the hit scores across all five images stay above 93 no matter how large the rotation — a direct consequence of "comparing direction, not gray value."

Two points are most worth savoring. First, **none of the distractors is falsely detected**: every image is given three slots (matchCount=3), yet the ring washer and toothed rack — parts with "the wrong shape" — never reach minScore=40. The gradient-direction list recognizes the wedge's unique combination of edge segments; the washer's arcs and the rack's square teeth simply do not match. Second, **the redrawn model contour fits precisely**: the gray thin line on each wedge in Figure 17.2 is the model's edges redrawn at the solved pose (including the rotation angle), wrapping the part's true boundary tightly — even the one in sample 2, flipped by 152°, is dead on.

Drag grayscale NCC in to do the same job, and it is immediately exposed. Using the upright wedge template (with no rotation search whatsoever) to compute correlation: at the upright instance in sample 1 the NCC is **1.000** (the template was cropped from there), but at the center of that rotated instance in sample 2 (the position given by shape matching) the NCC plunges to **0.585** — far below the customary 0.80 decision line. NCC's convolution window is nailed to one orientation, so the moment the workpiece turns it no longer aligns and the score collapses; shape matching makes angle an explicit search dimension and finds the part wherever it has turned. This is the most fundamental divide between NCC and shape matching: **the former recognizes only one pose, the latter searches the entire pose space.**

17.3 Occlusion Experiment

In the real samples the parts are well separated, with no ready-made occlusion scene, so a **synthetic mid-gray occluding bar** is overlaid on the wedge in sample 1 (simulating a carrier



(a) Sample 1: 0°

(b) Sample 2: −152°

(c) Sample 3: two (102°/89°)

(d) Sample 4: 1°

(e) Sample 5: −68°

(f) Template (reference)

Figure 17.2: All six wedges are hit: the black cross marks the solved center, and the gray thin line is the model edges redrawn at the solved pose. None of the distractors (ring, toothed rack) is falsely detected.

tape or gripper crossing the workpiece), with all other pixels kept real. The bar covers **36.0%** of the part's area (visible fraction 0.64). Searching again, shape matching still hits squarely: the pose locks at $(318, 237)$, 0° — identical to the un-occluded case — and the score drops from 99.8 to **76.5**, a degradation ratio of 0.77, the same order as the visible fraction of 0.64. The "score visible fraction" of Section 17.1 holds up broadly in the commercial implementation; that it runs a touch above 0.64 is because the bar happens to cross the middle of the part, while the wedge's directional features are more densely concentrated at the tip and the chamfer, which remain exposed.

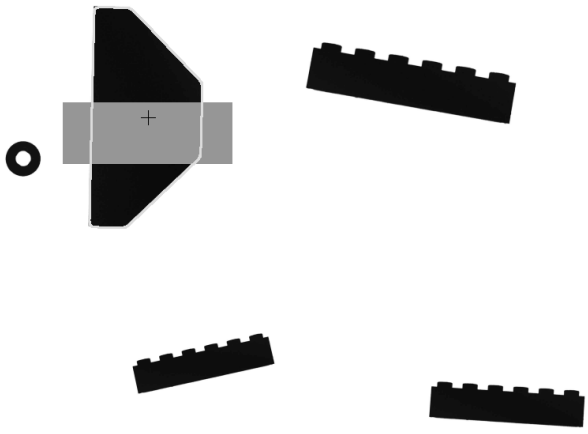


Figure 17.3: Occlusion experiment: a mid-gray bar overlaid on the real wedge (covering 36% of its area); shape matching still hits with a score of 76.5, with the pose identical to the un-occluded case. The light thin line is the redrawn model contour — it **runs straight across the occluding bar** into the invisible region, since the algorithm "knows" where the covered portion is.

From this follows the engineering reading of `minScore`: **it is the maximum occlusion fraction you are willing to tolerate.** An intact wedge scores 95–100; setting minScore to 40

tacitly permits about 60% of the contour to be missing, while setting it to 70 demands an essentially complete part. But occlusion resistance is not determined by minScore alone: the coarse-screening parameter `partialThreshold` requires a sufficient fraction of the model to be visible at the coarse pyramid levels, and it must be opened up along with the occlusion budget (0.4 here) — otherwise the occluded candidate is eliminated at the coarse level and never reaches fine scoring.

17.4 Scale Matching

Localization has a fourth degree of freedom. Once the camera-to-workpiece distance changes — a model changeover, a different pallet height, a refocused lens — the part's **scale** in the image changes, and a fixed-size template immediately fails to line up with the contour. The remedy is exactly as for angle: make scale an explicit search dimension too. At training time, poses are pre-generated on the angle $\times$ scale grid according to `startScale`/`scaleExtent`/`scaleStep`, and at search time every grid cell is scored.

The scale samples are designed for exactly this: the same set of parts photographed at three working distances, the parts shrinking from near to far. Take the **square connector** (a rectangular body with a protruding tab below, dark contour, non-rotationally-symmetric) from the nearest image (sample 1) as the model, and search all three images over a scale range of $[0.55, 1.20]$ (step 0.05):

Table 17.2: Measured scale matching: the connector shrinks with working distance, and the scale is recovered one by one

Scale sample	Solved scale	Solved angle	Score
1 (nearest, template source)	1.000	0°	97.8
2 (middle)	0.850	-2°	88.0
3 (farthest)	0.650	-2°	74.4

The three images solve at scales 1.000, 0.850, and 0.650 — the

part shrinks to 85% and then 65% of its original size as the camera pulls back, and the scale search recovers every step (Figure 17.4). The score falls with scale (97.8→88.0→74.4) because a smaller part has fewer effective edge points and the soft edges of a real photograph become harder to work with, but 74.4 is still far above minScore=25 and the localization is solid. The redrawn model contour, scaled by the solved factor, wraps each differently sized connector snugly.

(a) Sample 1: scale 1.000 (b) Sample 2: scale 0.850 (c) Sample 3: scale 0.650

Figure 17.4: Scale matching: the same connector localized at three working distances, solving at scales 1.000 / 0.850 / 0.650. The black thin line is the model contour redrawn after scaling by the solved factor, wrapping each differently sized part accurately.

17.5 Template Design

Not just any part in the samples can serve as a template, and this real dataset happens to lay three lessons out in the open.

First, **the template must break rotational symmetry.** Imagine modeling on that ring washer: a circle's contour rotated by any angle is still the same circle, so the vast majority of edge points in the model are "direction-consistent" with every angle; the score stays high but the reported angle is pure chance — the classic **rotational-symmetry trap**. This chapter chose the pentagonal wedge precisely because its vertical edge and chamfered corner contribute a large number of edge points with unique directions, so the angle can be locked firmly (the angle residuals of all six instances are small). The lesson

Scale matching fills in the fourth degree of freedom of the **pose**; it does not replace **calibration** (Chapter 5): when the working distance changes, the pixel-to-millimeter conversion changes too, and accurate dimensional measurement still requires recalibration. What scale matching solves is only "can we still find it after the working distance drifts, and by how much did it scale."

in one sentence: **the score only guarantees "the contour lined up," never "the pose is the unique solution"** — checking whether a template is (approximately) rotationally symmetric should be a fixed step before modeling.

Second, **features must be one of a kind.** The toothed rack in the samples is a row of repeating square teeth; were it the model, any equally spaced square-wave edge in the scene could claim a high score as an impostor. The wedge's combination of edge segments is unique among this set of parts, which is why every distractor is rejected. The template should be a **one-of-a-kind** combination of structures, and the threshold must sit above the highest score any distractor could earn.

Third, **the edges of a real photograph are soft.** The scale samples are real grayscale photos of metal parts, with edges only a few pixels wide and carrying reflections and noise. Box-average downsampling them directly drives the gradient magnitude below the contrast threshold, so that **not even the part's own image can be matched**; matching must be done at full resolution, with the pyramid limited to 2 levels (automatic layering wipes out the soft edges at the upper levels and the coarse screen fails). The silhouette samples have extremely high contrast, so none of these concerns arises — which itself shows that the "edge quality" of the template and samples directly dictates how the parameters must be set.

17.6 SciVision Implementation

First, a naming trap: the SDK's `SciSvOCV` is **not** contour matching — OCV is optical character verification, for character-defect verification. Contour matching and scaled contour matching are handled uniformly by `SCIMV::SciSvScaleShapeMatch`: `scaleExtent=0` gives pure contour matching, a nonzero range gives scale matching — one class covering both uses. Training and search:

```
SCIMV::SciSvScaleShapeMatch sm;
SciROI mask;                          // must be UNDEF: a GenRect1 rectangle raises 120001037
SciMatchModel model;
```

```
sm.CreateScaleImageModel(tmplImg, mask, /*pyramidLevel*/ 2,   // 2 levels for soft edges; -1 (a
    /*startAngle*/ -180, /*angleExtent*/ 360, /*angleStep*/ 1,
    /*startScale*/ 1.0f, /*scaleExtent*/ 0.0f, /*scaleStep*/ 0.05f,
    /*lowerContrast*/ -1, /*upperContrast*/ -1,        // edge hysteresis thresholds, -1 = auto
    /*filterCoefficient*/ 3.0f, /*filterLength*/ -1,   // training-side smoothing
    /*minContrast*/ -1, /*optimization*/ -1,
    /*polarity*/ 0, /*preGeneration*/ 1, &model);      // preGeneration must be 1

SciPointArray centers; SciVarArray angles, scales, scores;
sm.FindScaleImageModel(sceneImg, searchROI, model,
    -180, 360, 1.0f, 0.0f,
    /*minScore*/ 40, /*matchCount*/ 3, /*overlapRatio*/ 40,
    /*subpixel*/ 0, /*endLevel*/ 0, /*clutter*/ 0.0f,
    /*coarseScore*/ 35, /*partialThreshold*/ 0.5f, /*spreadWinSize*/ 2,
    &centers, &angles, &scales, &scores);
```

On the training side, the two triplets of angle and scale parameters mean the same; `lowerContrast`/`upperContrast` are the hysteresis double thresholds of the template's edge extraction, deciding which points enter the model list; `polarity` is the polarity switch of Section 17.1. On the search side, `minScore`/`matchCount`/`overlapRatio` control the threshold, the instance count, and deduplication; `GetScaleTemplateContours` retrieves the model's edge contours — this chapter's overlay figures are drawn exactly that way, redrawing them at the solved poses.

Pitfalls hit in our measurements, recorded faithfully. **First, mask must be passed as a default-constructed UNDEF ROI**: passing a `GenRect1` full-template rectangle raises error 120001037 outright — the same family temperament as feature matching and color matching. **Second, `preGeneration` must be 1** (full model pre-generation), or subsequent matching is unusable. **Third, `subpixel` can only be 0**: setting it to 1 (interpolated refinement) crashes outright with 0xC0000005; setting it to 2 (least-squares refinement) "drifts at high score" — a high score carrying an angle error of tens of degrees, all the more deceptive. **Fourth, soft-edged real photos require limiting the pyramid levels and matching at full resolution**: if this chapter's scale samples are box-average

downsampled as usual and searched with automatic layering (`pyramidLevel=-1`), not even the connector's own image can be matched ("match failure," 122411002); switching to full resolution and `pyramidLevel=2`, all three images solve their scales cleanly. Note also that `coarseScore` must be $\leq$ `minScore`, or the coarse levels screen out qualified candidates. The complete project is at `code/shape_matching/`.

Industry Case: Switching Locators for Oily Workpieces

A machining shop needed pick-up localization on workpieces smeared with cutting-fluid oil and started with gray-value matching: the oil film made surface gray values undulate with the lighting angle, scores swung by as much as 30 points within a single day, and the threshold could not be tuned. Switching to shape matching stabilized things immediately — what the oil changes is the gray-value distribution, while the workpiece's **contour** does not budge, and gradient-direction similarity is naturally immune. But the first version simply boxed the entire workpiece, sweeping all of the internal machining texture into the edge-point list: modeling and matching were both slow, and scores jittered whenever the oil locally wiped out some texture. The second version kept only the outer contour and two key locating holes, and both speed and stability passed. Two lessons: shape matching's robustness comes from "comparing only the reliable edges," and stuffing unreliable detail into the model is self-sabotage; template design matters as much as algorithm selection.

17.7 Summary

- Shape matching abstracts the template into a list of **edge points + gradient directions**; the score $s = \frac{1}{n}\sum \cos \Delta\theta_i$ is the average direction agreement: the model is a list of points with geometric coordinates, so it can search explicitly over angle and scale; gradient direction is unchanged by gain/offset and even by locally monotonic nonlinear mappings; missing points only dilute the score without disturbing the rest — searcha-

bility, illumination robustness, and occlusion tolerance all spring from the same source.

- **Rotation robustness**: a single pentagonal-wedge template finds all six instances across five real samples, with angles spanning -152° to $+102^\circ$ and scores all ≥ 93.7, and not one ring or toothed-rack distractor falsely detected; at the same rotated instance grayscale NCC is only 0.585 (< 0.80), confirming that NCC does not resist rotation.
- **Under occlusion, score visible fraction**: a synthetic bar covers 36% of the real wedge, the score goes 99.8→76.5 (degradation ratio 0.77), and the pose is still exact; `minScore` is the maximum tolerable occlusion fraction, and coarse-screening parameters such as `partialThreshold` must be opened up along with the occlusion budget.
- **Scale**, the fourth dimension, is opened by `scaleExtent`; as the real connector shrinks with working distance, the three images solve at scales 1.000 / 0.850 / 0.650 in turn. Scale matching completes the pose; it does not replace calibration.
- On the SDK side, remember a few things: pass mask as UNDEF, preGeneration=1, subpixel only 0, coarseScore $\leq$ minScore; **soft-edged real photos must be matched at full resolution with the pyramid limited to 2 levels**, or not even the part's own image can be matched.

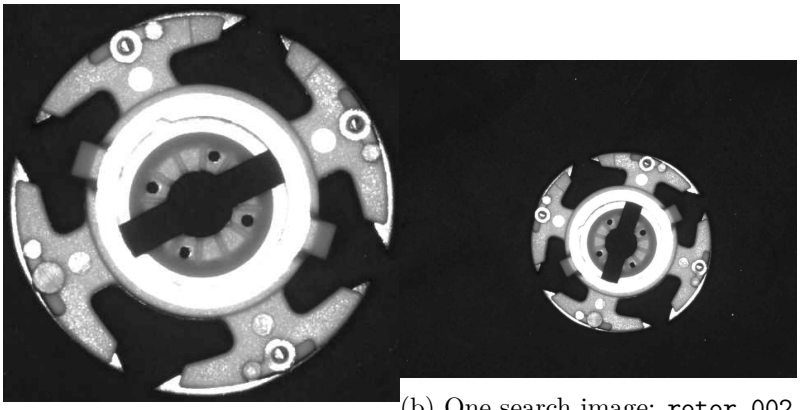
Shape matching solves the hardest form of "where is the part"; when the target is not a rigid contour but a color blob or texture features, feature matching and color matching remain as options (Chapter 18). Edge-based shape matching has two further classic threads: Borgefors's hierarchical chamfer matching measures contour similarity via a distance transform (Borgefors 1988), and Belongie et al.'s shape contexts describe point-set shape with distribution histograms and solve the correspondence (Belongie, Malik, and Puzicha 2002). The principles and engineering details of shape matching are treated most systematically in the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

18 Feature and Color Matching

The matching methods of the previous two chapters share one trait: they are all **dense** — gray-value matching (Chapter 16) compares an entire template pixel by pixel, shape matching (Chapter 17) compares an entire contour point by point, and pose (angle, scale) must be searched as explicit dimensions step by step, with cost growing as the range widens. This chapter introduces two kinds of matching that take a different road altogether. **Feature matching** compresses an image into a few dozen **keypoints**; what gets compared is no longer a patch of pixels but a sparse point set — when a part may appear at an arbitrary angle, or even at a different scale, its cost barely grows with the pose range. **Color matching** abandons geometry entirely: in a sorting task like "is this a red cap or a yellow cap," what matters is not where the part is or how far it has rotated, but **what color it is** — classifying by color statistics is more direct than any geometric matching.

The feature-matching part uses a set of **real industrial sample images**: the sample sheets from Smart3's "feature matching" locating recipe — a five-armed rotor/sprocket casting, imaged under dark-field illumination as 1000×800 8-bit grayscale. The same physical part appears in four images at different positions and angles, which is exactly where feature matching shines. We crop a part-sized square out of one of them (`rotor_001`, the part roughly centered and high) to use as the **template** (Figure 18.1a), and search for it in the other three (`rotor_002/003/004`). The part's surface is covered with sandblast casting texture and dotted with through-holes, slots, and arm tips, so corners abound; yet it is also near five-fold rotationally symmetric — which plants a real pitfall for matching, detailed below. The color-matching part keeps synthetic data: this "feature matching" sample set is single-channel grayscale

and cannot drive color matching, so to fully present the use and chromatic behavior of `SciSvColorMatch`, the color experiment uses a synthetic red/green/blue/yellow bottle-cap sorting scene, as noted in Section 18.4.



(a) Template: the rotor part cropped from `rotor_001` (480 × 480)

(b) One search image: `rotor_002`, the part in another pose

Figure 18.1: Test data for the feature-matching experiment (real samples, Smart3 feature-matching recipe). (a) Template: a 480 × 480 square cropped around the brightness centroid of the part in `rotor_001` — the sandblast texture, through-holes, slots, and five arm tips provide hundreds of corners; (b) one 1000 × 800 search image: the same part on a dark field at a different position and rotation, the background near-black and featureless.

18.1 Keypoints and Descriptors

The first step of feature matching is to pick out a small number of "distinctive" points from the image. What makes a point distinctive? A pixel inside a flat region looks the same wherever its neighborhood is shifted — there is nothing to recognize; a point on an edge keeps a nearly unchanged neighborhood while sliding along the edge, so only one direction can be pinned down. Only a **corner** — a point where the gray value changes

sharply along two independent directions — has a neighborhood that differs under every translation and can be uniquely "recognized." This is the same intuition as corners serving as salient contour points in Chapter 25: the detector scans the image for locations where the local "cornerness" is strongest and exceeds a threshold, and takes those as keypoints.

Position alone is not enough — we must also be able to **recognize** the same point in another image. For this, a **descriptor** is computed over each keypoint's neighborhood: the gray-value or gradient distribution of the neighborhood is encoded into a feature vector that serves as the point's "fingerprint." The point pairs whose descriptors are closest across the two images become candidate **correspondences**. For the same corner to keep a similar fingerprint after rotation, the descriptor first estimates the dominant orientation of the neighborhood and rotates the neighborhood into a canonical orientation before encoding — this is where **rotation invariance** comes from: the angle is not found by exhaustive search but is "digested" by the descriptor.

This is precisely where sparse matching turns the cost tables. Dense template matching must traverse the pose space: a grid of position $\times$ angle $\times$ scale, with a full-template comparison at every grid point — double the angle range and the cost doubles. Feature matching splits the pipeline into three steps: detect keypoints (once per image, independent of pose), compare a few dozen descriptors, and finally solve for the pose from the candidate correspondences — the pose space is never enumerated from start to finish. The candidate correspondences inevitably contain misrecognized **outliers**, so the pose is solved with RANSAC: repeatedly draw a minimal point set at random, fit a rigid (or similarity) transform, and count how many correspondences agree with that transform — the **inliers**; the transform with the most inliers wins, and the pose is refined by least squares over all inliers.

There is no free lunch, of course: feature matching requires the template to have **enough detectable corners**. A smooth, textureless part (a disk, a plain plate) yields hardly any keypoints, and the model simply cannot be built — this chapter's rotor casting is the opposite case, with corners everywhere across its

We already used RANSAC when fitting lines in Chapter 14: amid data contaminated by outliers, "draw minimal samples at random, vote for a consensus" is far more robust than least squares. There the outliers were stray edge points; here they are misrecognized descriptor correspondences — the recipe is exactly the same.

A quick cost ledger: covering 360° on this chapter's sample with gray-value matching at a 1° step would require ~360 template poses and scanning each one; feature matching only detects keypoints once (about 410 per search image), then finds correspondences among descriptor pairs on the order of 335 $\times$ ~410 — and however much wider the angle range gets, this bill does not change.

sandblast surface and its holes, slots, and arm tips, a "friendly" subject for feature matching. But it brings a different, very real trouble: near five-fold rotational symmetry. On a geometrically self-similar part, descriptors on different arms look alike, and the pose can easily settle on a "displaced" consensus — exactly the engineering pitfall the next section confronts.

18.2 Feature Matching Experiment

The template comes from `rotor_001`: we first threshold the part by brightness to find its centroid ($(533.3, 492.6)$) and crop a 480×480 square around it (Figure 18.1a). Training on the template with the STABLE detector yields **335 model feature points** (Figure 18.2): they blanket the sandblast surface, the through-hole and slot edges, and the arm tips — the places of strongest cornerness. The same model finds about **410 scene keypoints** in each of the three 1000×800 search images (420 / 414 / 406).

The real samples carry no ground-truth pose, so we validate the matches with two mutually independent lines of evidence. **The first is a known-answer self-match**: searching for the model back in its own source image `rotor_001` should return it to the template origin. The measured location is $(533.00, 493.00)$, angle $0.000°$, scale 1.0000 — an error of **0.00 px** against the template origin $(533, 493)$, which zeroes out the entire crop-model-search chain. **The second is a cross-check against the intensity centroid**: the rotor is nearly centrally symmetric, so its brightness centroid is a stable physical reference entirely unrelated to feature matching; if the solved center agrees with it, the two corroborate each other.

The matching results for the three images (Figure 18.3) are clean, every one hitting on the first try at the **strictest matcher threshold, 60**:

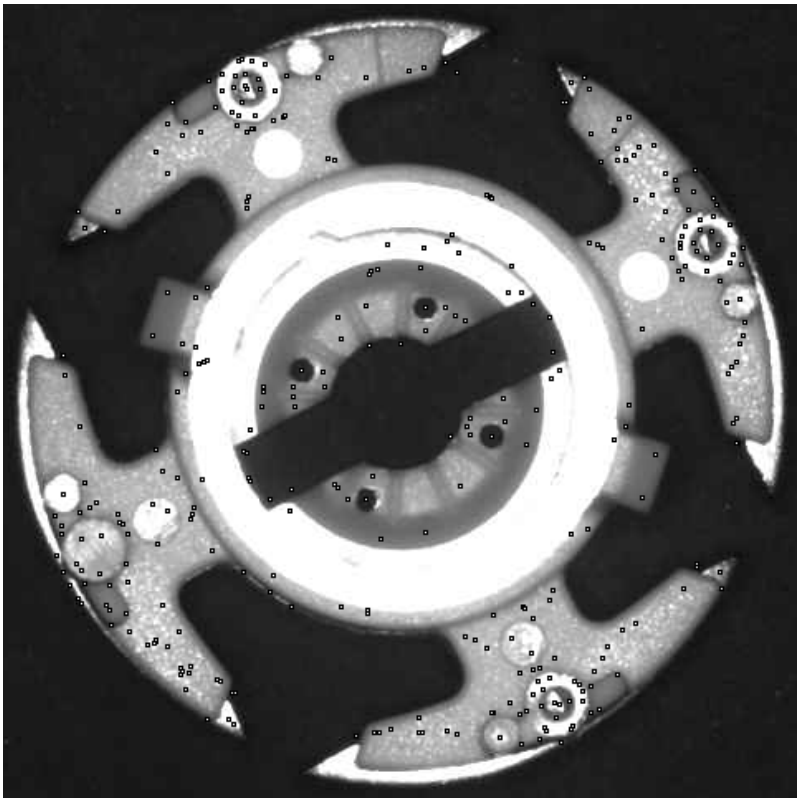


Figure 18.2: The 335 model feature points obtained by training (marked as black squares with white centers): blanketing the sandblast texture, hole and slot edges, and arm tips — the corner-rich locations.

Table 18.1: Pose-solving results for the three search images (angles in image coordinates, y pointing down)

Search image	Solved center	Angle	Scale	Intensity centroid	Center error
`rotor_002`	$(558.72, 458.24)$	$-44.97°$	0.9996	$(558.6, 457.8)$	**0.43 px**
`rotor_003`	$(701.58, 239.17)$	$-149.99°$	0.9995	$(701.1, 239.0)$	**0.51 px**
`rotor_004`	$(308.59, 558.20)$	$-104.98°$	1.0000	$(308.6, 557.8)$	**0.37 px**

The center errors are all on the order of **0.5 px**, and the scales without exception fall within 0.05% of 1.000 — squarely consistent with the physical fact of "same camera, same working distance, unscaled part," strong corroborating evidence that the results are sound. Note that we never told the matcher the angle range: rotations from $-45°$ to $-150°$ were absorbed naturally by the descriptors and RANSAC. And one subtle but important point: the solved angles ($-45°$, $-150°$) are not integer multiples of the five-fold symmetry step $72°$ — which means the match was not "talked into" some symmetry-equivalent solution by the rotational symmetry, but anchored a genuine absolute orientation through the casting's **unique sandblast texture**.

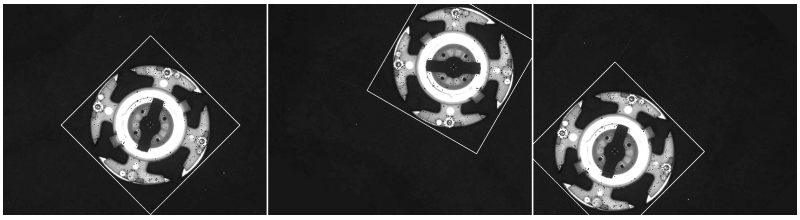


Figure 18.3: Feature-matching results (three search images side by side): white boxes show the solved poses, crosses mark the centers, and black squares with white centers are the model feature points mapped by the solved pose. The same part appears at a different position and rotation in each image, all framed accurately, with the mapped model points landing precisely on the part structure.

This experiment exposed an engineering reality textbooks almost never write down but that is enough to wreck a deploy-

ment on the line: **the choice of descriptor**. The first attempt used the STABLE detector **plus the STABLE descriptor** (the manual's "robust" default pairing); the self-match was perfect and `rotor_003` localized to 3.6 px, but `rotor_002` and `rotor_004` were both **badly displaced** — centers off by **183 px and 52 px**, scales drifting to 1.06 and 0.78, and the matcher threshold having to be relaxed all the way to 10 before any result appeared. The root cause is exactly the near five-fold symmetry planted in the previous section: the STABLE descriptor lacks the discriminating power to tell the self-similar arms apart, and RANSAC voted its consensus onto a rotation-displaced solution. Switching to the **FAST descriptor** made all three images hit at the strict threshold 60, with center errors down to 0.5 px and scales back at 1.00 — same detector, same set of keypoints, only the descriptor changed, and the results were worlds apart. The lesson: on geometrically self-similar parts, the descriptor's discriminating power matters more than the detector's point count; "default parameters" are not necessarily tuned for your part, and **a quick empirical sweep over the competing parameters** often finds the answer faster than reading the manual.

One more SDK semantic is worth recording: even with `matchCount` set to 2, `FindFeatureImageModel` **returns only 1 instance per call** — RANSAC picks the single transform with the most inliers, and in its eyes the second-best instance's correspondences are just a pile of outliers. Each image here happens to contain exactly one part, so one call per image suffices; but if an image held several copies of the same part, the application layer must iterate "find one → erase it from the image → find again."

Finally, the source of sparse matching's robustness deserves a word: the model points in Figure 18.3 map precisely onto each part because **the pose is determined by the geometric consistency of the point group**, not the accuracy of any single point. The detector only recognizes "cornerness," so individual candidate correspondences inevitably connect to the wrong arm or to a bright background speck; what rules them out is the hard constraint that all 335 model points must fall into place **at once**. The robustness of sparse matching lies not

in the accuracy of any single point, but in the agreement of the group.

18.3 Representing and Matching Color

Switch scenes: red, green, blue, and yellow bottle caps stream past on a conveyor, and the task is to sort them. Geometry is completely irrelevant here — the caps are all round — what matters is only the color. The most naive approach treats a region's mean color as a three-dimensional vector (R, G, B), computes the Euclidean distance to each reference color, and lets the nearest one win.

There is a mine buried in this approach: **RGB couples brightness and chromaticity together**. Take the same red cap and dim the lighting by thirty percent — the three components of (R, G, B) shrink in nearly equal proportion, the color vector slides along the ray toward the origin, and its Euclidean distance to the reference color grows, even though to the human eye it is "still that same red." On a production line, lamp tubes age, voltage fluctuates, and workpieces sit at varying distances from the lights — brightness is never a constant; sorting by raw RGB distance amounts to betting the classification threshold on the stability of the illumination.

The fix is to **divide the brightness out** of the color, keeping only the "ratio of the colors" — this is **chromaticity**. The simplest chromaticity coordinates are normalized RGB:

$$r = \frac{R}{R+G+B}, \qquad g = \frac{G}{R+G+B},$$

(with b given by $r + g + b = 1$). Multiply all three components by a common gain and the numerator and denominator scale together — (r, g) does not budge. A linear brightness change is canceled exactly, the same idea by which NCC cancels the illumination gain through normalization (Chapter 16), replayed in color space. The separation of hue H, saturation S, and value V in HSV space (already used for color thresholding in Chapter 7) is another realization of the same idea, with equivalent engineering effect.

The color matcher is used **one model per class**: train one color model for each reference color, score the region under test with every model, and the highest score gives the predicted class. Whether its internal color representation is chromaticity-like is something the manual may never say outright — but we can interrogate it with an experiment.

The geometric picture: in RGB space, points of "same color, different brightness" line up on a ray through the origin; chromaticity normalization projects each ray onto a single point on the plane $R + G + B = \text{const}$. Classification is done on this plane, with the brightness dimension folded away entirely.

18.4 Color Matching Experiment

This section uses **synthetic data**: the Smart3 samples used for feature matching above are single-channel grayscale and cannot carry color information, so to fully demonstrate the use of `SciSvColorMatch` and probe its internal color representation, we construct a color-patch sorting scene. The reference samples are four bottle-cap color patches (red/green/blue/yellow, each with ± 10 uniform noise added per pixel per channel, Figure 18.4). The test set is a 3×4 grid (Figure 18.5): the four columns correspond to the four colors, and the three rows multiply the caps' brightness by $0.8\times$, $1.0\times$, and $1.2\times$ respectively — simulating $\pm 20\%$ fluctuations in illuminance and reflectance.

Figure 18.4: The four reference color patches: red, green, blue, and yellow bottle caps, one color model trained for each.

The classification result is clean: **12/12 correct**, with true-class scores falling between 81.7 and 92.9 and an average margin over the strongest wrong class of about **40 points** — not a narrow win but a rout.

The real point of this experiment, however, is not the accuracy but the **brightness sweep**. Averaging the true-class scores by row:

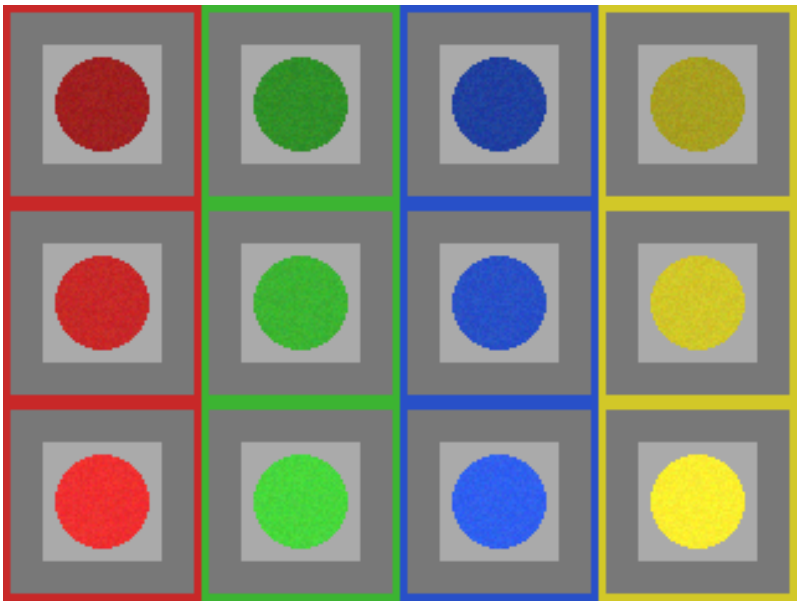


Figure 18.5: Color-matching results: each patch is outlined in the reference color of its predicted class (row brightness from top to bottom $0.8\times$ /$1.0\times$ /$1.2\times$). All 12 patches are classified correctly; not one carries the diagonal line that would mark a misclassification.

Table 18.2: True-class score versus brightness (mean over the 4 patches in each row)

Brightness	0.8×	1.0×	1.2×
Mean true-class score	91.0	89.9	84.1

With brightness swinging 20% up and down, the score curve is nearly flat — a maximum drop of only **5.8 points**. If the matcher used raw RGB distance internally, a ±20% brightness change would push the color vector quite far from the reference, and the score should slide substantially; the flat curve points to a different conclusion: **this SDK's color matching behaves close to a chromaticity space, not raw RGB distance**. The slight dip in the 1.2× row also has a sound explanation: the yellow cap's R-channel reference value is 210; multiplied by 1.2 it reaches 252, already brushing the ceiling of 255, and with ±10 noise added it gets clipped — saturation genuinely shifts the chromaticity, so the points docked are docked for good reason.

One methodological detail deserves to be spelled out. We also ran two hand-written baselines: classification by **raw RGB distance** on the disk means, and by **chromaticity distance** — both also scored 12/12. This tells us the four-color sorting task is simply too easy: red, green, blue, and yellow sit far apart in hue, and ±20% brightness is not enough to make raw RGB stumble, so **accuracy as a metric has zero discriminating power on the question "is the SDK chromaticity-based."** What can separate the two hypotheses is the shape of the score curve: the raw-RGB hypothesis predicts scores sliding visibly with brightness, the chromaticity hypothesis predicts a flat curve — and the measurements side with the latter. When designing an experiment, make the competing explanations issue **different predictions**; otherwise even the prettiest numbers are mere self-congratulation.

18.5 SciVision Implementation

Feature matching is handled by `SCIMV::SciSvFeatureMatch`, in two steps — training and search:

```
SCIMV::SciSvFeatureMatch fm;
SciROI mask;                                  // default-constructed = UNDEF, meaning the whole ima
SciMatchModel model;
fm.CreateFeatureImageModel(tmplImg, mask,
        SCI_FEATUREMATCH_DETECTOR_STABLE,     // detectorMethod: 0=STABLE 1=FAST 2=EDGEPOINT
        SCI_FEATUREMATCH_DESCRIPTOR_FAST,     // descriptorMethod: FAST far beats STABLE on this
        /*sampleStep*/ 1,                     // sampling step: 1/2/4 - larger means sparser poi
        /*scaleRange*/ 0,                     // scale levels 0..3, 0 = no scale expansion
        /*detectorThreshold*/ 30,             // corner-response threshold [1,255]; higher means
        &model);
SciFeaturePointArray pts;
fm.GetFeatureModelPoints(model, &pts);        // 335 model feature points in this example

SciPointArray centers; SciVarArray angles, scales;
fm.FindFeatureImageModel(sceneImg, roi, model,
        SCI_FEATUREMATCH_MATCHER_STABLE,
        /*matcherThreshold*/ 60,              // match-quality floor [1,100]; the strict 60 hits
        /*matchCount*/ 1,                     // note: returns only 1 instance however large it
        &centers, &angles, &scales);
```

On the training side the most decisive choice is not the threshold but the `descriptorMethod`: on this chapter's near-symmetric rotor the STABLE descriptor cannot tell the similar arms apart and solves the pose onto a displaced consensus, while the FAST descriptor is stable (see Section 18.2). `detectorThreshold` and `sampleStep` control point density and speed; `scaleRange` only needs to be opened up when the part's distance may vary (e.g., the camera height is not fixed). On the search side, `matcherThreshold` is the inlier-rate gate discussed in Section 18.2 — with the descriptor chosen well, all three images here hit at the strictest setting of 60. The returned `angles` follow the mathematical positive direction (counterclockwise positive); to draw the pose in image coordinates with the y-axis pointing down, negate

and convert — the same convention pitfall as in gray-value matching (Chapter 16).

Color matching is handled by `SCIMV::SciSvColorMatch`:

```
SCIMV::SciSvColorMatch cm;
SciROI cmask;                              // likewise must be UNDEF
SciMatchModel cmodel;
cm.CreateColorImageModel(refImg, cmask,
        /*pyramidLevel*/ -1,               // -1 = choose the number of pyramid levels automati
        SCI_COLOR_SENSITIVITY_HIGH,        // color sensitivity 0..2
        &cmodel);

SciPointArray ctr; SciVarArray ang, score;
cm.FindColorImageModel(gridImg, cellROI, cmodel,
        /*minScore*/ 1, /*matchCount*/ 1, /*overlapRatio*/ 50,
        /*startAngle*/ 0, /*angleExtent*/ 10, /*angleStep*/ 1,  // extent must be >= 10
        &ctr, &ang, &score);
```

For classification, call `FindColorImageModel` once per model on each ROI under test and take the highest score; a model returning "no match" (122407002) on a region of a different color is normal — just treat it as a score of 0.

Five pitfalls hit in practice, recorded as they happened:

- **The descriptor choice makes or breaks the match.** With the same detector and the same set of keypoints, the STABLE descriptor mislocalized `rotor_002/004` on the near-symmetric rotor (centers off by 183 px / 52 px, scales drifting to 1.06 / 0.78), while the FAST descriptor hit all three images at the strict threshold 60 with errors ≤ 0.5 px. On geometrically self-similar parts, always test both descriptors empirically.
- **The mask must be passed as UNDEF.** The `mask` parameter of both `Create*Model` calls must be a **default-constructed SciROI** denoting the whole image; passing a `GenRect1` full-image rectangle makes feature matching report "insufficient feature points" (122406005), while color matching trains "successfully" but then fails at matching (122407002).

- **`FindFeatureImageModel` has single-instance semantics.** However large `matchCount` is, only 1 instance comes back; multiple instances must be handled at the application level with the "find → erase → find again" iteration.
- **`GetFeatureImagePoints` demands a rectangular ROI instead.** Exactly the opposite of Create: give it an UNDEF ROI and it reports 122406101; you must pass a full-image rectangle `GenRect1(0,0)-(W,H)` (bottom-right corner exclusive). Never assume ROI semantics are uniform across APIs of the same family.
- **`angleExtent` of `FindColorImageModel` must not be 0.** Even though the patches are plainly unrotated, `angleExtent=0` returns 122407002 **intermittently**; passing ≥ 10 everywhere made it stable. Intermittent failures are harder to track down than steady ones — when you meet one, suspect boundary-value parameters first.

The complete runnable project is located at `code/feature_color_matching/`.

Industry Case: Lighting Drift in Terminal Color Sorting

A connector factory sorted terminals by housing color; the first version classified with fixed threshold boxes in RGB space. After months of operation the misclassification rate crept up month by month, and the investigation traced it to aging fluorescent tubes on the shop floor causing a combined drift in color temperature and illuminance: the RGB threshold boxes had been calibrated under the lighting of acceptance day, and once the lamps changed, the color vectors translated wholesale out of the boxes. The fix came in two steps: on the feature side, switch classification to chromaticity coordinates (normalized RGB), isolating the brightness dimension up front; in addition, fix a white reference tile in a corner of the field of view and recompute the **white balance** gains from its measured RGB every shift, correcting the residual color-temperature drift. After the retrofit the misclassification rate fell back and stabilized. Two lessons: a color task must isolate the "brightness channel" out of its features; and the system needs a **"white" that can be recalibrated over time** — the illumination never stops changing, and an environment that changes demands a reference that can change with it.

18.6 Summary

- **Feature matching = keypoints + descriptors + geometric consistency**: detect sparse keypoints of strong cornerness, build candidate correspondences with rotation-normalized descriptors, and let RANSAC vote out the pose amid outliers — the pose space is never enumerated, so large angles and scale changes cost far less than dense matching. This chapter uses real rotor samples: 335 model points against about 410 scene points, the same part appearing at a different pose in three images, all with center errors ≤ 0.5 px and scales fixed at 1.000 (the self-match error of 0.00 px zeroes the chain).
- **The robustness of sparse matching lies in the agreement of the group, not the accuracy of any single point**: individual keypoints connect to the wrong arm or to a background speck, and only the geometric consistency of all 335 model points falling into place at once keeps them out of the pose; the price is that the template must be corner-rich — smooth, textureless parts need not apply.
- **Engineering reality: the descriptor choice is the decider for geometrically self-similar parts**. On the near-symmetric rotor the STABLE descriptor mislocalized (off by 50-180 px), and only the FAST descriptor held all three images steady at the strict threshold 60; in addition, `FindFeatureImageModel` returns one instance per call, so multiple instances require "find-erase-refind."
- **Classify color by chromaticity, not raw RGB**: chromaticity coordinates like the normalized $r = R/(R+G+B)$ divide brightness out of color, which is what keeps classification stable under lighting fluctuations; measured on this SDK, the true-class score dropped only 5.8 points across a $\pm 20\%$ brightness sweep — behavior close to a chromaticity space.
- **Design experiments with discriminating power**: even the raw-RGB baseline aces the four-color sorting task, so accuracy cannot separate the hypotheses; what can is the shape of the score curve versus brightness

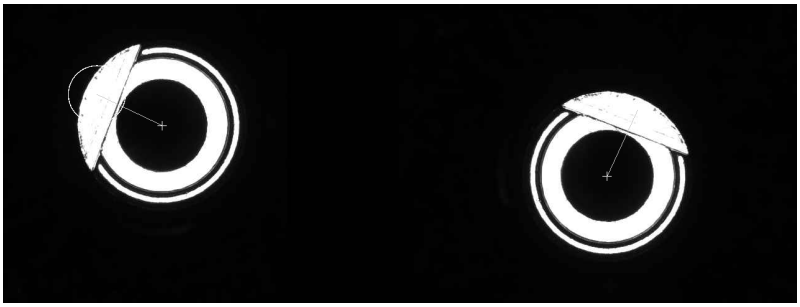
— only when competing hypotheses issue different predictions do the data get a say.

The three pillars of sparse feature matching each have a foundational reference: the Harris–Stephens corner detector (Harris and Stephens 1988), Lowe's SIFT, which pushed detection and description to scale invariance (Lowe 2004), and Rublee et al.'s ORB, which greatly accelerates matters with a binary descriptor (Rublee et al. 2011). Feature points, descriptors, and color-based recognition are treated more systematically in the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

19 ROI Generation and Fixturing

The previous chapters all solved the problem of "finding the part": template matching (Chapter 16), shape matching (Chapter 17), and geometric primitive locating (Chapter 14) each work their own magic, but all ultimately hand over one position and one angle. On a production line, though, the real problem often only begins there — what happens after the part is found? A typical inspection station measures a dozen or even dozens of items on one part: hole diameters, edge distances, characters, defects… Every measurement item has its own ROI (region of interest), and the part arrives at a different position every time. Should every measurement item carry its own locating step? The standard answer on industrial lines is a far more elegant pattern: **locate once, benefit everywhere** — use a single locate to obtain the part's current pose, then "carry" all the measurement ROIs to the part's new position with one and the same transform. This mechanism is called **fixturing**. This chapter walks it through on a set of real sample images: the part under test is a rigid piece consisting of an illuminated ring plus a curved tab (Figure 19.1a, from the Smart3 "ROI correction sample recipe"), whose tab surface carries scratches and is precisely the object to inspect; Figure 19.1b shows the same part moved elsewhere in the frame and rotated as a whole by about 90°. The white cross is the located ring center, and the thin line points to the tab. The inspection ROI (white circle) sits on the tab in the reference image — all the work ahead amounts to making that circular ROI catch up with the tab, which has moved 234 px.

This sample set contains 7 images, all the same rigid part imaged at different incoming poses: the ring's inner-hole radius is stable at **74.0 px** across all 7, the distance from the tab centroid to the ring center is stable at **116 px**, and the bright-pixel



(a) Reference scene (sample 001): the ring center (cross) plus the direction line to the tab; the inspection ROI (white circle) sits on the tab

(b) Current scene (sample 002): the same part translated by (+78.9, +81.0) px and rotated by about 90°; the white cross is the located ring center

Figure 19.1: The reference pose and the current pose. The inspection ROI is defined on the reference scene; once the part moves, an ROI that stays put misses completely (the tab moved 234 px).

count is constant at about 29565 — these invariants are the fingerprint of "one and the same rigid body". The part appears in three orientations: 001/006 (tab at upper left; the two are in fact a duplicate of the same image), 002–005 (tab rotated to the upper right; the four are pure translations of each other), and 007 (tab rotated to the lower left).

19.1 ROI Types and Generation

ROIs have appeared again and again in the preceding chapters; here is a systematic inventory. SciVision's `SciROI` generates the various shapes through a family of `Gen*` methods: the **rectangle** (`GenRect1`, axis-aligned) is the most general — thresholding, filtering, and almost every other region operation uses it; the **rotated rectangle** frames tilted edges or character lines; the **circle** (`GenCircle`) naturally matches circular features such as holes, shafts, and solder joints, and this chapter's inspection ROI is a circle sitting on the tab — the radial search lines of Chapter 14 are likewise laid out inside a circular ROI; the **torus** (`GenTorus`) constrains the search band between an

inner and an outer radius, exactly right for measuring sealing rings and gaskets; the **polygon** hugs irregular contours. One engineering fact is worth recalling: the bottom-right corner of `GenRect1` is an **exclusive** endpoint — a full-image ROI should be passed as (W, H) rather than (W−1, H−1), or the outermost row and column will go unprocessed (Chapter 7).

Since the ROI must move with the part, one has to ask: what does each ROI type become under a rigid transform? The center moves, the radius stays — **a circle remains a circle**, and this chapter's circular inspection ROI is still a circle after a fixturing with an angle (measured below); the same goes for the torus. An axis-aligned rectangle, once rotated, is **no longer axis-aligned** — it becomes a rotated rectangle, which is exactly why rotated-rectangle ROIs are everywhere in fixturing pipelines: the upright rectangles drawn casually on the reference image all acquire an orientation after one fixturing with an angle. A polygon transforms vertex by vertex, its shape unchanged. In other words, all of these ROI types are closed under rigid transforms (under an affine transform a circle would become an ellipse, but fixturing never needs a transform that general), so there is no geometric obstacle whatsoever to ROIs following the part.

An ROI has two identities, and beginners often see only the first. The first identity is **bounding the computation**: a 640×480 image has three hundred thousand pixels, while a circular ROI of radius 46 around the tab has fewer than seven thousand — the algorithm runs only inside the ROI, a one-to-two-orders-of-magnitude difference in speed. The second identity matters more: the ROI binds the **measurement semantics** — "inside this circular ROI there should be the tab, and we are checking its surface for scratches". The algorithm's parameters (expected grey level, polarity, thresholds) are all configured according to "what is inside the ROI"; once the ROI no longer covers the feature it is supposed to cover, those parameters lose their premise entirely, and the algorithm either fails or — more dangerously — computes a plausible-looking result on the wrong content. This chapter's experiment demonstrates the latter case on purpose.

19.2 Pose and the Rigid Transform

A **pose** is the union of "position + orientation": the pose of a rigid planar part is fully determined by three quantities, the coordinates of a base point $\mathbf{p} = (x, y)$ and an angle θ. The base point can be the template origin reported by a matching algorithm, or, as in this chapter, the center of a circular feature; the angle is taken along some directional reference. This chapter uses the **center of the ring's inner hole** as the base point (the translation origin) and the **direction from that center to the tab centroid** as the angle — the center supplies the translation, the tab supplies the rotation, and together they lock down the pose of the whole part. Let a point $\mathbf{f}$ in the part's local coordinate frame (say the tab centroid) have an image position determined by the pose:

$$\mathbf{x} = R(\theta)\,\mathbf{f} + \mathbf{p}, \qquad R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

The reference pose $(\mathbf{p}_r, \theta_r)$ and the current pose $(\mathbf{p}_c, \theta_c)$ give the same point two image coordinates $\mathbf{x}_r$ and $\mathbf{x}_c$; eliminating $\mathbf{f}$ yields the **rigid transform** from reference image coordinates to current image coordinates:

$$\mathbf{x}_c = R(\Delta\theta)\,(\mathbf{x}_r - \mathbf{p}_r) + \mathbf{p}_c, \qquad \Delta\theta = \theta_c - \theta_r.$$

Written as a homogeneous matrix this is the familiar 3×3 form from Chapter 2: a rotation part $R(\Delta\theta)$ and a translation part $\mathbf{p}_c - R(\Delta\theta)\,\mathbf{p}_r$. The entire mathematics of fixturing is this single matrix: it holds for **any** point on the reference image, so one locate can drive arbitrarily many ROIs.

Hidden here is a very practical property, one that this chapter's "coarse" angle estimate happens to confirm. Our angle does not come from a high-precision edge fit; it is the centroid direction of the bright pixels in the tab region — an estimator with a **systematic bias** that can be pulled off by several degrees by uneven illumination or the distribution of scratches. But as long as this bias is approximately constant in the **part frame** (same tab, same imaging), write it as $\hat{\mathbf{p}} = \mathbf{p} + R(\theta)\,\mathbf{b}$ and

substitute into the relative transform, expanding about a true feature point $\mathbf{x}_r = R(\theta_r)\mathbf{f} + \mathbf{p}_r$ on the reference image:

$$R(\Delta\theta)\,(\mathbf{x}_r - \hat{\mathbf{p}}_r) + \hat{\mathbf{p}}_c = R(\theta_c)\,\mathbf{f} - R(\theta_c)\,\mathbf{b} + \mathbf{p}_c + R(\theta_c)\,\mathbf{b} = \mathbf{x}_c.$$

The two $R(\theta_c)\,\mathbf{b}$ terms **cancel exactly** — as long as the bias moves with the part, it has no effect on the fixturing whatsoever. The experiment bears this out: even though the angle estimate itself is a coarse centroid direction, because the reference and current images use the **same** estimator, the residual between the fixtured ROI center and the independently measured tab centroid stays $\leq$ **0.27 px** throughout (worst case sample 004, 0.266 px). This is fixturing's most easily overlooked teaching point: fixturing uses a **relative** transform, so a systematic bias in the locating feature is "wrong by the same amount on both sides" and cancels on subtraction; what truly eats precision is the random jitter of the angle and the part's own non-rigidity.

The word "fixturing" comes from the machining fixture: a fixture locks the workpiece in a fixed pose so the tool path needs no sensing of the workpiece. Vision fixturing does the opposite — the workpiece is placed freely, and the "tool path" (the ROI) adapts to the workpiece — a **software fixture**.

The cancellation requires the bias is constant in the part frame. If the bias comes from factors in the **image frame** such as illumination direction or lens distortion, it does not rotate with the part and will not cancel; another reason the locating feature should be a high-contrast, well-isolated structure (such as this chapter's illuminated inner hole).

19.3 The Fixturing Experiment

The complete chain has four steps: **locate → compute the fixturing matrix → transform the ROI → measure**. Locating splits into two branches: the ring center is found from the inner-hole edge — scanning radially over angles to find the dark→bright transition gives the inner-hole edge points, and a least-squares circle fit on them converges the inner-hole radius to 74.0 px on all 7 images; the angle is then taken as the direction from the center toward the tab centroid (the centroid of bright pixels at radii in [134, 210]). Note that the locating extent and the measurement ROI play different roles: locating scans the large-scale structure of the whole part (a ring over a hundred pixels across), large enough to lock the part at any incoming position; the inspection ROI is the small, accurate one that must be carried by fixturing. With sample 001 as the reference, the locate result is ring center (255.56, 194.54) and tab direction −154.5°; the inspection ROI is then defined on the tab centroid (150.93, 144.67) measured on the reference image

(a circle of radius 46) — fixturing carries "the position seen on the reference image", so keeping the ROI definition consistent with the locate result is the engineering convention.

The second step hands the two poses to `SciAxisTransform::Generate2DAlignMatrix`, yielding the 3×3 fixturing matrix from the reference pose to the current pose. Compared element by element with the rigid matrix computed by hand from the previous section's formula, the maximum difference is below 10^{-3} — what the SDK does is exactly that mathematics. A self-check with `TransformPointAndAngle` follows: pass the reference center through the matrix, and the residual to the current center is 0.0000 px, fully consistent.

We initially tried SciVision's `SciSvEllipseLocator` to fit the inner-hole circle, but it selects edges unstably on this sample set: it returns 123501012 outright on 001/006, and on 002–005 it locks onto the outer-ring edge (radius about 87 px) rather than the inner hole (74 px), with the center off by about 24 px. A center inconsistent from image to image would utterly corrupt the relative pose, so we switched to a deterministic "radial edge + circle fit", whose inner-hole radius is stable at 74.0 px across the whole set.

The third step, `AlignROI`, transforms the inspection ROI (the radius-46 px circle hung on the tab in the reference image) to the current pose. A rigid transform does not change the shape, so a circular ROI stays a circle after transformation. Running this pipeline over all 7 samples, the recovered poses and alignment residuals are listed in Table 19.1: 002–005 are rotations of about +90° about the reference plus various translations, 007 is about −90°, and 006 coincides with 001 (the duplicate image, recovered as the identity transform). For every image the fixtured ROI center agrees with the independently measured tab centroid down to sub-pixel — **the worst residual is only 0.266 px**.

Table 19.1: Recovered poses and fixturing residuals for the 7 real samples (reference = 001). The "bright-pixel fraction" is the fraction of pixels with grey ≥ 100 inside the inspection ROI, reflecting whether the ROI covers the illuminated tab.

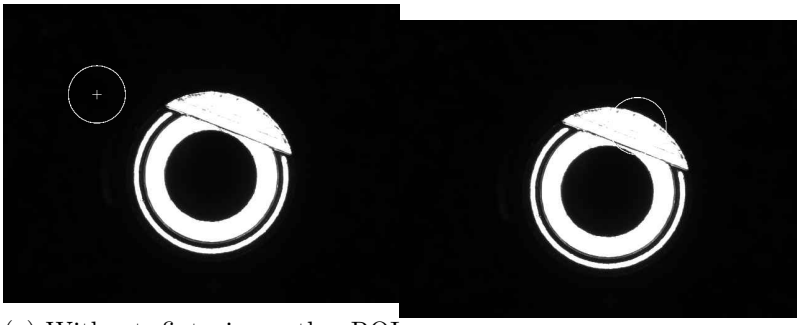
Sample	Recovered translation (px)	Recovered rotation $\Delta\theta$	ROI→tab residual	Bright-pixel fraction (fixtured/unfixtured)
001	(0.0, 0.0)	0.00°	0.000 px	0.718 / 0.718
002	(+78.9, +81.0)	+89.36°	0.164 px	0.718 / **0.000**

Sample	Recovered translation (px)	Recovered rotation $\Delta\theta$	ROI→tab residual	Bright-pixel fraction (fixtured/unfixtured)
003	(+128.9, +81.0)	+89.84°	0.097 px	0.718 / **0.000**
004	(+128.9, +131.0)	+89.73°	0.266 px	0.718 / **0.000**
005	(+128.9, −19.0)	+89.80°	0.202 px	0.719 / **0.000**
006	(0.0, 0.0)	0.00°	0.000 px	0.718 / 0.718
007	(−51.0, −21.1)	−89.81°	0.119 px	0.720 / 0.388

The core comparison is in the last column. Take sample 002: the tab moved 234 px from its reference position; after fixturing the circular ROI has a bright-pixel fraction of 0.718 (the tab fills the ROI), whereas the ROI left in place has a bright-pixel fraction of **0.000** — it lands on the pitch-black background in the upper-left of the frame, not covering a single illuminated pixel. Figure 19.2 compares the two visually.

The fourth step runs the inspection inside the fixtured ROI — this chapter uses the "bright-pixel fraction" as the observable for whether the tab is in place (a real line would inspect for scratches or measure the contour inside this ROI): the fixtured ROI gives 0.718, exactly matching the 0.718 of the same ROI on the reference image, so fixturing eats essentially none of the inspection budget.

What truly warrants caution is the result on the unfixtured side. Intuitively, with the ROI off by over two hundred pixels, the inspection should simply fail and raise an alarm; but if the inspection algorithm only checks the return code, it will **return normally** on that black background — bright-pixel fraction 0.000, "no scratch found", a seemingly perfect pass. This is the classic **silent failure**: the pipeline raises no error, the fields are filled in, yet the values were computed on the wrong content.



(a) Without fixturing: the ROI stays at the reference position, landing on the black background, bright-pixel fraction 0.000

(b) After fixturing: the ROI sits squarely on the tab, bright-pixel fraction 0.718; the cross is the fixtured ROI center

Figure 19.2: Unfixtured versus fixtured ROI in the same current scene (sample 002). After fixturing the ROI center is only 0.164 px from the true tab position.

It is far more dangerous than an outright error — an error stops the line, while a silent pass lets a bad part through. The engineering countermeasure is to guard the measurement with **structural thresholds**: the amount of valid signal inside the ROI (e.g. a bright-pixel fraction that is too low), whether the feature exists at all, the deviation from an expected model — any one out of bounds marks the measurement invalid, never trusting the return code alone. Here the gulf between 0.000 and 0.718 is a ready-made criterion.

19.4 Two Routes: Move the ROI or Move the Image

Fixturing has a second route, also the "tempting wrong turn" discussed in Chapter 10: instead of moving the ROI, transform the entire current image back to the reference pose by the inverse of the fixturing matrix, then measure with the original ROI as usual. This chapter implements that route with a hand-written bilinear inverse mapping (the same as the hand-written rotation of Chapter 10); the result is Figure 19.3: after sample 002 is rotated back, the part nearly reproduces the layout of

reference image 001, and the original ROI again sits on the tab.

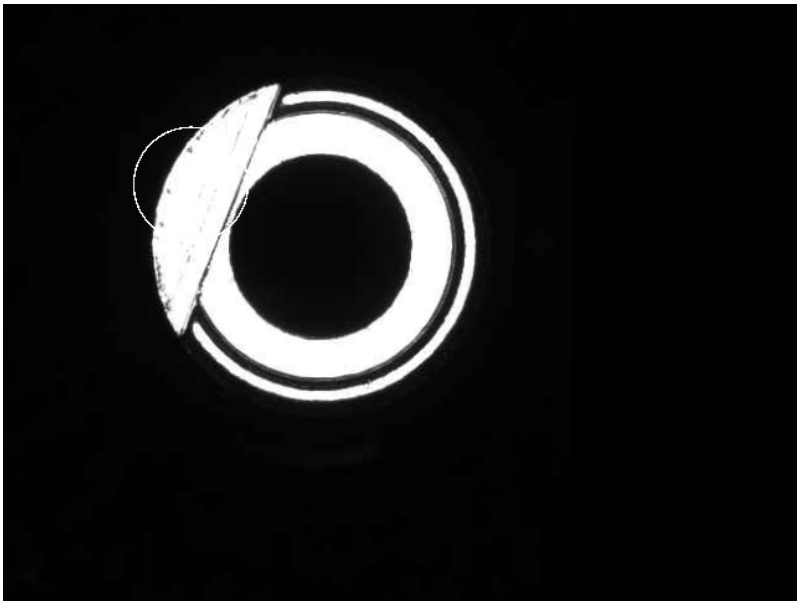


Figure 19.3: After the current image (002) is rotated back to the reference pose by the fixturing matrix, the original ROI measures. The part returns to 001's layout, and the original ROI again covers the tab.

The two routes give almost equal bright-pixel fractions inside the fixtured ROI: moving the ROI gives 0.718, rotating the image gives **0.722** — mathematically they are the same transform, the difference being only interpolation noise (0.004). But the cost is wholly unequal: moving the ROI transforms only a few geometric parameters (two numbers for the center, one for the radius), at negligible cost; moving the image resamples all three hundred thousand pixels once, and bilinear interpolation lays a layer of smoothing over the edges, degrading the repeatability of edge localization (the case in Chapter 10 quantified this degradation). The conclusion matches that chapter: **if you can move the ROI, do not move the image**. Moving the image is worthwhile only in two situations: the algorithm accepts only axis-aligned ROIs and cannot express an oriented inspection region; or several stations and several algorithms must share one reference coordinate frame, where a single uni-

fied rotation is more controllable than each stage fixturing on its own.

19.5 SciVision Implementation

The heart of fixturing is two interfaces of `SciAxisTransform`:

```
SCIMV::SciAxisTransform xform;
SciMatrix M;
SciPoint refPt(poseRef.cx, poseRef.cy), curPt(poseCur.cx, poseCur.cy);
long rc = xform.Generate2DAlignMatrix(refPt, curPt,
    (float)(-poseRef.deg), (float)(-poseCur.deg), &M);   // negate the angles, see below

SciROI fixedROI;
rc = xform.AlignROI(inspROI, M, &fixedROI);              // a circular ROI stays a circle
```

The four inputs to `Generate2DAlignMatrix` are, in order, the reference base point `refPt`, the current base point `curPt`, the reference angle, and the current angle; it outputs the 3×3 matrix `M` from the reference pose to the current pose. `AlignROI` passes any `SciROI` through the matrix to obtain the fixtured ROI. To verify the matrix is correct, use `TransformPointAndAngle`, which transforms a point and an angle together, well suited to the self-check "the reference center should land on the current center".

This chapter hit two pitfalls on real samples; both are recorded faithfully. **The first pitfall most deserves underlining: the sign of the angle.** `Generate2DAlignMatrix` adopts the convention **visual counter-clockwise is positive**; but the pixel coordinate system has the y axis pointing down, so the angle computed by `atan2(dy, dx)` is positive clockwise in the visual sense — the two differ by a sign. Pass the `atan2` result straight in and the rotation direction reverses, sending the fixtured ROI off by hundreds of pixels, worse than no fixturing at all. The correct practice is to pass `-poseRef.deg` and `-poseCur.deg`. The handedness issue of coordinate frames was emphasized in Chapter 2; this is its most painful appearance.

The second pitfall is in the locating step: `SciSvEllipseLocator` **selects edges unstably** on this set of illuminated-ring samples — it returns 123501012 outright on two of the images, and on the other four it locks onto the outer-ring edge (radius about 87 px) instead of the target inner hole (74 px), with the center off by about 24 px. Fixturing consumes the **difference** of two locates, so once the locating source drifts and loses self-consistency from image to image, even the most accurate matrix is fed garbage input. This chapter therefore switched to a deterministic "radial dark→bright edge points + least-squares circle fit" for the inner-hole center, and all 7 images converge to the same radius 74.0 px, which is what makes self-consistency possible. This lesson outweighs any single-point accuracy: **the precision ceiling of a fixturing system is set by the repeatability of locating, not by the absolute accuracy of a single frame.** The complete project that produces all figures and numbers in this chapter is at `code/roi_and_fixturing/`.

Industry Case: Many Measurements Share One Locate

A connector inspection station had 23 measurement items: pin pitch, plastic-shell edge distance, latch contour… At first each item ran its own template-matching locate, badly overrunning the cycle time, and because each item's locating error was independent, cross-item dimensions like "pin to shell edge" jittered noticeably. The redesign located the whole connector once by shape matching and transformed all 23 measurement ROIs through one and the same fixturing matrix: cycle time dropped by about 60%, and more importantly all items shared one coordinate datum — the relations among items were no longer polluted by their individual locating errors. The price is that locating quality becomes a **global single point of dependency**: once locating fails or drifts, all 23 items go down together. This is exactly the lesson this chapter's samples teach — the **repeatability** of locating is the lifeline of a fixturing system. In deployment a score threshold and failure retries (changing parameters, changing the search region) were added to locating, and the trend of the locating score across product batches was brought into monitoring.

19.6 Summary

- **The standard pattern of fixturing is "locate once, benefit everywhere"**: one locate gives the current pose, and one rigid-transform matrix drives all measurement ROIs to follow the part — across this chapter's 7 real samples, the residual between the fixtured ROI and the tab is everywhere ≤ 0.27 px, while the unfixtured ROI misses outright (the tab moved 234 px, and the ROI's bright-pixel fraction fell from 0.718 to 0.000).
- **Fixturing uses a relative transform, so the systematic bias of the locating feature cancels automatically**: this chapter's angle estimate is merely the tab's centroid direction (biased), but the reference and current images use the same estimator, so the bias zeroes out in the subtraction of the $R(\theta_c)\mathbf{b}$ terms and the residual stays sub-pixel.
- **A silent failure is more dangerous than an error**: the unfixtured ROI lands on the black background, and an inspection that checks only the return code would "normally" report "no defect". Measurements must be guarded by structural thresholds such as signal amount, feature existence, and parameter plausibility, never trusting the return code alone — here the gulf between 0.000 and 0.718 is the ready-made criterion.
- **If you can move the ROI, do not move the image**: moving the ROI and rotating the image agree to within 0.004 in the inspection result, but the latter pays one full-frame resample and interpolation smoothing. Move the image only when the algorithm cannot take an oriented ROI, or when multiple stations must share a datum frame.
- **The precision ceiling of fixturing is the repeatability of locating, not single-frame absolute accuracy**: `SciSvEllipseLocator` selecting edges inconsistently across images (off by 24 px) corrupts the relative pose; only after switching to a self-consistent inner-hole circle fit (radius constant at 74.0 px) did fixturing become stable. The angle sign is another pitfall — `Generate2DAlignMatrix` takes visual counter-clockwise

as positive, and the pixel `atan2` result must be negated before being passed in.

For a systematic treatment of pose estimation and rigid and affine alignment in industrial measurement, see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018); for the broader algorithmic background of alignment and image registration, see Szeliski's textbook (Szeliski 2022).

Part V

Measurement

This part covers the core techniques of industrial measurement: caliper tools and subpixel edge localization, geometric measurement of distances and angles, and quantitative measurement of intensity, color, and gaps.

20 Caliper Measurement and Subpixel Localization

With this chapter we formally enter the territory of measurement. The preceding chapters answered "what is the target and where is it"; the measurement chapters must answer "how wide, how far, how round is it" — converting pixels into millimeters. The calibration of Chapter 5 provides the scale factor, but "measuring millimeters with pixels" still has a last mile: a micron-resolution system that must make demanding dimensional judgments needs edge positions measured to far less than one pixel. The standard tool for walking this last mile is the **caliper**: a directed one-dimensional measuring device — given an ROI, a search direction, and an edge polarity, it extracts a gray profile along the search direction, finds one edge or a pair of edges, and directly outputs subpixel positions and width. Using a real industrial sample — a metallographic side-section of a Micro-USB connector — this chapter quantifies the caliper's three core questions one by one: where subpixel accuracy comes from, what determines repeatability, and why precision and accuracy must be verified separately.

The object of measurement is shown in Figure 20.1: on a 2592×1944 grayscale image, against a dark background, lies the side cross-section of a Micro-USB connector, with a row of contact pins standing inside the bright metal shell. Each pin is a bright vertical metal bar (gray about 240), flanked on both sides by dark background (gray about 70) — sharp contrast, steep boundaries. This is exactly the caliper's home turf: **search horizontally to measure pin width and the spacing between pins**. Connector-assembly yield depends heavily on the geometric consistency of the pins; pins that are too wide, too narrow, or misaligned cause abnormal insertion force or poor contact, so pin width and pitch are the critical

dimensions of such parts. This chapter takes two adjacent pins (denoted pin A and pin B) as the measurement targets.

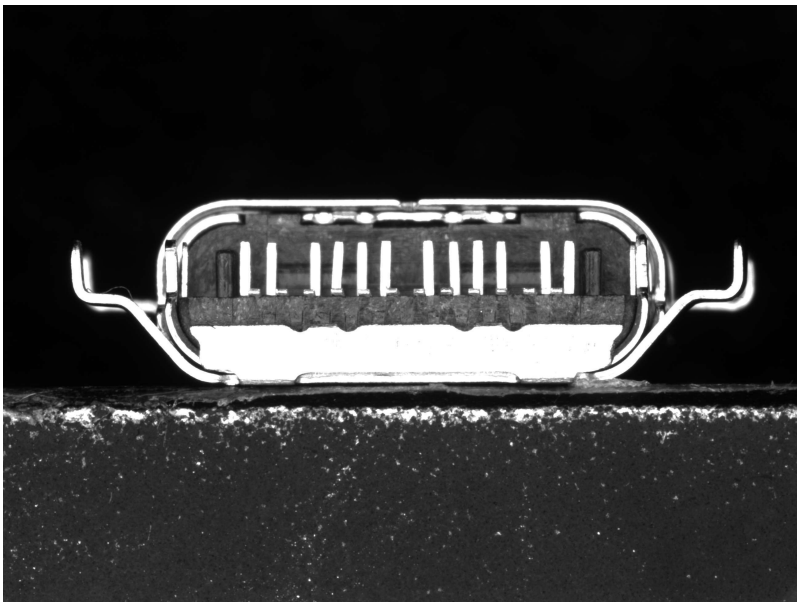


Figure 20.1: The real measurement object: a Micro-USB connector side cross-section (2592×1944 grayscale). Against a dark background, a bright metal shell and a row of vertical contact pins; pins are about 33 px wide with about 73 px pitch. This chapter applies caliper measurement to two adjacent pins in the central region.

20.1 How the Caliper Works

The caliper shares its roots with the search-line machinery of Chapter 14: take a gray profile along the search direction inside the ROI, suppress noise by projection averaging in the perpendicular direction, take the first derivative of the profile, and apply parabolic interpolation at the derivative extremum to obtain the subpixel edge position. If an edge-point extractor outputs "a string of points, left for you to fit", the caliper differentiates itself in three ways: first, it is natively oriented toward the **edge pair** — it locates two edges and directly gives

the distance between them, i.e. the width; second, when multiple candidate edges appear on the profile, it **selects edges** by scoring rules such as strength and expected distance, giving it built-in immunity to false edges caused by oil stains and scratches; third, its output is the measurement value itself, with no post-processing required. Its relationship to Chapter 13 can be stated in one sentence: edge detection answers "where in the image are there edges", while the caliper is the one-dimensional specialization for "we already know roughly where the edge is and need to measure it accurately" — the search range collapses from the full 2D image to a single line, and both speed and accuracy benefit.

Let us first lay out the measurement results (Figure 20.2). SciVision's SDK caliper gives pin A a left edge of x=1003.42, a right edge of x=1036.15, **width 32.73 px**; pin B a left edge of x=1076.68, a right edge of x=1108.92, **width 32.24 px**. The center-to-center spacing (pitch) of the two pins is **73.02 px**, and the dark gap from pin A's right edge to pin B's left edge is **40.53 px**. These are direct readings in the pixel domain; if the system is calibrated at s m/pixel, the pin pitch is $73.02\,s$ m. Note that this SDK's caliper fit line is always vertical (detailed in Section 20.5), so it measures the **horizontal** distance; the pins are nearly vertical (the measured tilt via `LinePointsLocator` is only 0.16°), so the difference between horizontal and perpendicular width appears only in the second decimal place.

The "caliper" takes its name from the mechanical vernier caliper: two jaws grip the workpiece to read out the width. The software caliper is its virtual counterpart — the ROI is the body, and the two edges are the contact points of the jaws. A vernier caliper subdivides the scale with its vernier; a software caliper subdivides the pixel with parabolic interpolation — the idea runs in the same vein.

20.2 Where Subpixel Accuracy Comes From

The pixel grid has a spacing of 1; on what basis can we measure an edge to within 0.1 pixel? The answer lies in the physical nature of edges: an edge in a real image is never a mathematical step. The lens point-spread function smears the sharp physical boundary into a gradient, and the photosite further smooths it by integrating the light intensity over its sensitive area (Chapter 3) — by the time it reaches the digital image, the edge is already a gray ramp spanning one to several pixels. Figure 20.3 magnifies pin A's left edge 8×: between the dark background

This chapter's SDK output uses the pixel **corner** coordinate convention (integer pixel boundaries are integers, pixel centers are x.5): pin A's left edge x=1003.42 lies between columns 1003 and 1004. When comparing subpixel results across libraries, align the coordinate conventions first, or an "error" of 0.5 px will appear out of nowhere.

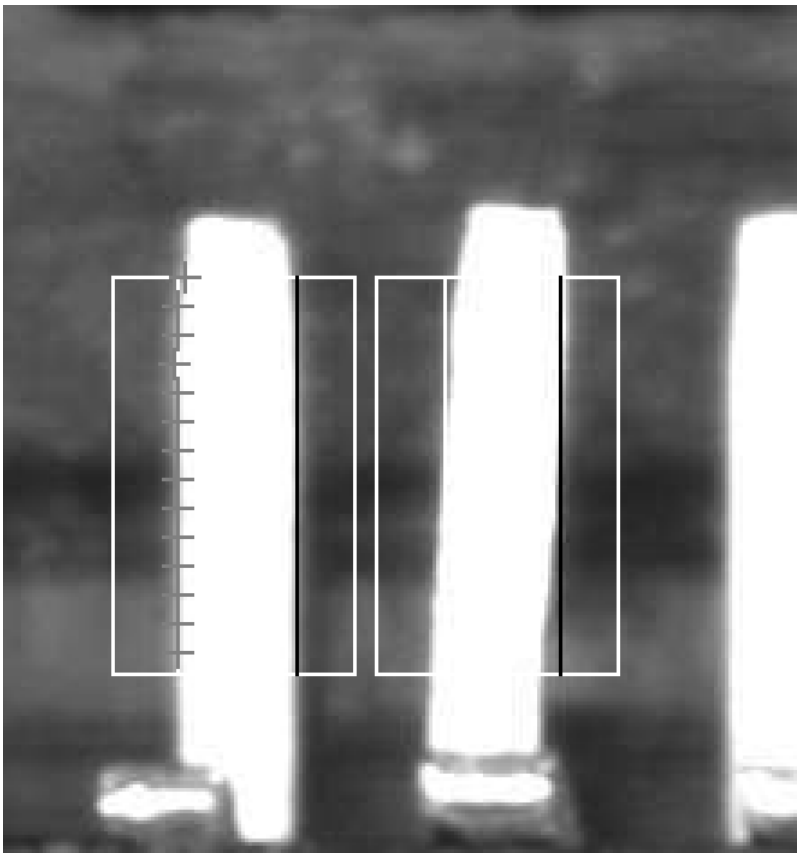


Figure 20.2: Caliper measurement results (central two-pin region, 3× zoom). White boxes are the two caliper ROIs; each pin's left edge is marked with a white line and its right edge with a black line (the SDK-fitted vertical edge lines); the gray crosses on pin A's left edge are the subpixel edge points extracted per search row.

and the bright pin there is no single cut but a smoothly transitioning gray band, with the "half-gray" pixels plainly visible. **The gradient is precisely where the information lives**: the true edge position continuously determines the gray value of every pixel in the transition band — move the edge right by 0.1 px and the transition pixels' gray levels change proportionally. What subpixel localization does is decode the continuous position from the specific values of these few half-gray pixels.

The decoding tool is the parabolic interpolation already used in Chapter 14 (its least-squares roots are in Chapter 2): fit a parabola to the discrete extremum of the profile's first-derivative magnitude and its neighbors g_{-1}, g_0, g_{+1}; the vertex offset $\delta = \dfrac{g_{-1} - g_{+1}}{2(g_{-1} - 2g_0 + g_{+1})}$ is the subpixel correction. Figure 20.4 spreads this process out on pin A's left edge: the gray dots are the raw gray samples of a single row (y=845), jittered by sensor noise; the black polyline is the profile after projection averaging over 111 rows, with a clean ramp outline; the vertical dashed line is the subpixel edge position obtained by interpolating the averaged profile, x=1002.91.

The interpolation formula itself is closed-form; the quality of the accuracy depends on how clean the profile fed to it is. Extracting the subpixel edge row by row on pin A's left edge, the positions of 111 search rows have a mean of 1002.96 px and a standard deviation of **0.37 px** — this 0.37 px contains both sensor noise and the real workpiece geometry (see Section 20.4). The caliper jointly fits these 20 search rows into a single vertical line, and the jitter of the single-point estimate is lowered by $\sqrt{N}$ averaging, so the final output is far more stable than any single row. This is precisely the subject Section 20.3 will quantify.

20.3 Repeatability: The Lifeline of Measurement

One of the quantities industrial measurement cares about most is: how much do results scatter when the same workpiece is measured repeatedly — **repeatability**. Precision means "measures

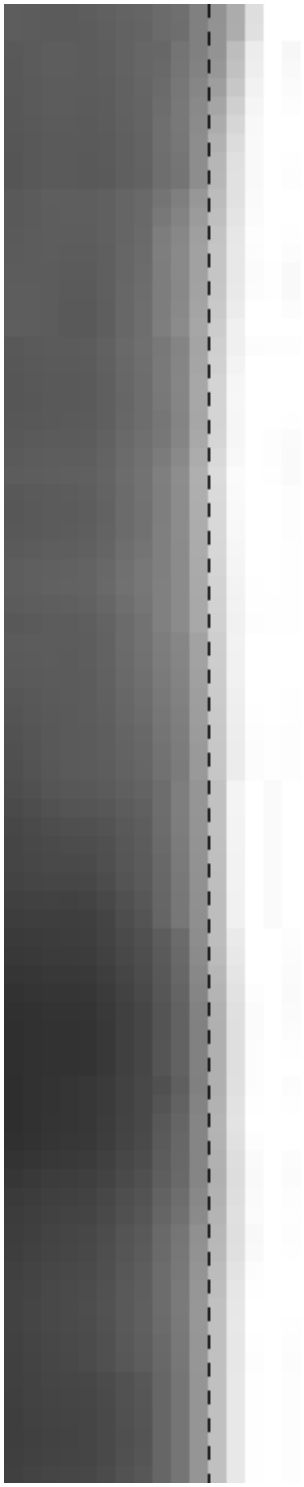


Figure 20.3: 8× magnification of pin A's left edge. Between the dark background (left) and the bright pin (right) is a smooth gray transition band rather than a hard step — the subpixel information is carried precisely in these "half-gray" pixels. The vertical dashed line is the subpixel edge position obtained by parabolic interpolation (x=1002.96).

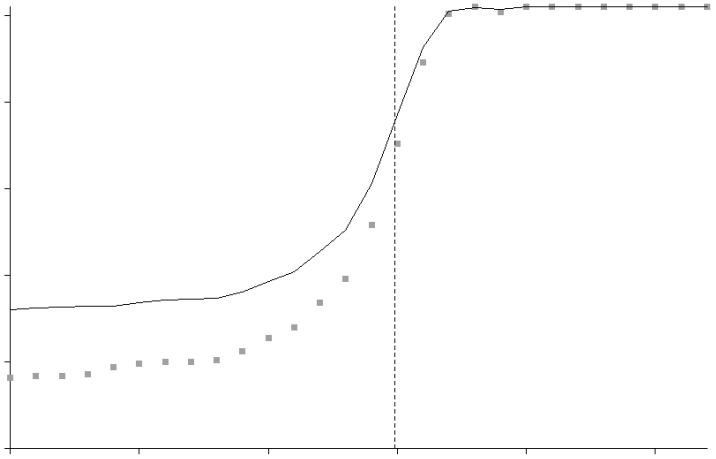


Figure 20.4: Gray profile of pin A's left edge. Gray dots: raw samples of a single row (y=845), containing sensor noise; black line: profile after projection averaging over 111 rows; vertical dashed line: the subpixel edge position obtained by parabolic interpolation of the averaged profile (x=1002.91).

steadily" (small variance), accuracy means "measures correctly" (mean close to ground truth); production-line decision logic is built on "the fluctuation of the measurement is far smaller than the tolerance band", and a system with poor repeatability — even with an unbiased mean — will misjudge good parts as defective and put defective parts into the good pile.

A real sample is a single image with no ensemble of noise for a Monte Carlo, so we reproduce repeatability via **ROI nudging**: translate the caliper ROI up and down along the pin direction by ±8 px (17 positions in total, with a slightly different set of sampled search rows each time), and tabulate the scatter of four measured quantities:

Accurate vs. steady: accuracy bias can be corrected by calibration and standard parts, but random scatter can only be lowered by improving the measurement system itself. The rule of thumb in gauge capability analysis (GR&R): the measurement system's repeatability (often taken as ±3 scatter) should not exceed 1/10 of the tolerance band, or measurement error will significantly erode the credibility of the judgment.

Quantity	Single measurement (px)	Repeatability (px, 17 ROI nudges)
Pin A width	32.73	**0.073**
Pin B width	32.24	**0.052**
Pin pitch (center)	73.02	0.373
Gap width	40.53	0.362

The repeatability of both pin widths is in the range 0.05–0.07 px: converted to a typical system at a few m/pixel, this amounts to sub-0.5 m single-measurement scatter. The most worthwhile comparison is with the single-row scatter — Section 20.2 measured the per-search-row subpixel edge scatter as 0.37 px, while the caliper jointly fits 20 search rows and the width repeatability drops to 0.073 px, about 1/5 of the single-row scatter; this is exactly the $\sqrt{N}$ law of Chapter 2: $0.37/\sqrt{20} \approx 0.083$, on the same order as the measured 0.073. **Multi-search-row projection/fitting is repeatability amplification that is nearly free** — lengthen the ROI along the edge, add search rows, and the variance of the position estimate falls proportionally, with the ceiling set by how perpendicular the edge is to the search lines.

Why is the of pin pitch and gap (0.37, 0.36 px) nearly five times that of a single pin width? Because each is obtained by subtracting the edges of **two different pins, two independent ROIs**, and the fitting errors of the two ends are no longer as highly correlated as the left and right edges of the same pin,

so the scatter is amplified by approximately independent superposition. This is the general rule for composite-quantity measurement: error grows with the number and independence of the participating edges, and measuring pitch is more "delicate" than measuring a single width.

20.4 Precision Is Not Accuracy

The 0.07 px repeatability measured in the previous section is splendid, but it only answers "how steady is the measurement", not "how correct is it". This is the most fundamental difference between a real sample and a synthetic scene: a synthetic image carries its own ground truth and lets you compute bias directly; this connector sample has no certified ground-truth pin width whatsoever, so **we can only give precision, not accuracy**. When reporting the number 32.73 px, we must be honest — its repeatability is 0.07 px, but how far it lies from the true physical width cannot be judged from this image. The only way to complete this link on a production line is to place a **standard gauge block** whose dimension is certified by a metrology institute, and use it to anchor the pixel reading to physical units and correct the system bias — this is the same thing Chapter 5 stresses with "calibration gives the scale, and calibration expires".

Real data also reveals a detail invisible on synthetic images: that 0.37 px of per-row scatter is **not all noise**. The subpixel positions of pin A's left edge across 111 rows range from 1002.42 to 1004.59, spanning about 2 px — this is not random jitter but a slight taper and curvature of the stamped metal pin's edge itself, real **workpiece geometry**. The caliper's vertical-line fit "presses" this slightly curved edge into a representative straight line, and the 0.37 px mixes two components: sensor noise and part shape. This is an important engineering lesson: **do not treat all scatter as noise** — sometimes the scatter hides real shape information about the part. If you care whether the pin is bent, you must dig this residual out of the fit and analyze it, rather than smearing it away with ever more averaging.

A synthetic scene can compute bias because the ground truth is written by us; a real sample measures an "unknown part", and without ground truth there is only precision. This is not a defect but the norm: the vast majority of production-line measurements have no per-part ground truth, relying instead on the combination of standard-part calibration and repeatability monitoring to guarantee credibility.

From this we obtain the chapter's practical doctrine: **when evaluating an algorithm with real parts, report precision and accuracy separately**. Repeatability can be measured directly from ROI nudging of a single image, or from repeated captures of the same workpiece; accuracy, however, requires an external standard part. Conflating the two — speaking of "accuracy" with a ground-truth-free sample, or passing off the bias of an idealized synthetic image as real performance — leads to conclusions disconnected from the production-line reality.

20.5 SciVision Implementation

The caliper is provided by `SCIMV::SciSvCaliper`, but the current state of this module needs honest disclosure. The old interface described in manual section 7.1 is blanked out in the header with an `#if 0`; what is actually usable is the current `Caliper()`. It empirically has several behaviors at odds with the documentation: of the polarity parameter, only `polarity1=2` can find an edge, and its measured semantics are "from dark to bright along the search direction" — exactly opposite to the header comment; passing 0 or 1 always raises error 122504007; the edge-pair mode (`edge2Idx>=1`) never outputs a width (`widthArr` is always 0); the differential caliper `mode=1` always fails; and the fitted edge line is always vertical (angle output always 90°). The core function of "the caliper measuring width" is implemented in engineering with a **two single-edge-call** workaround: inside an ROI containing only a single pin, search once with `direction=2` (left to right) and once with `direction=3` (right to left) (the dark→bright polarity hits one side of the pin each time), and the difference in x of the two vertical fit lines is the horizontal width:

```
EdgeDirection dir;
dir.direction1 = direction;  // 2=left-to-right; 3=right-to-left (the two calls hit the two pi
dir.polarity1  = 2;          // empirically the only usable: dark->bright along search (opposi
EdgeFilter filter;
filter.searchLineCount = 20; // 20 search rows inside the ROI
filter.edgeWidth = 3; filter.projectWidth = 1;       // mode=0 already projects over the whole
```

```
filter.sensitivity = 30; filter.strengthThresh = 30; filter.strengthLimit = 255;
EdgeScoring scoring;
scoring.maxResultNum = 5; scoring.expectDist = 34; scoring.sortMethod = 0;
SCIMV::SciSvCaliper cal;
long rc = cal.Caliper(src, roi, region, /*mode caliper*/0, dir, filter, /*trend*/0,
                      /*pattern specify edge*/0, /*edge1Idx*/1, /*edge2Idx single edge*/0, sco
                      &avg, &widthArr, &scoreArr, &mxi, &mni, &pos1, &ang1Arr, &pos2, &ang2Arr
double edgeX = 0.5 * (pos1[0].x + pos1[1].x);        // fit line is always vertical, both endpo
// pin width = max(e2,e3) - min(e2,e3); pin center = mean of the two edges; pitch = difference
```

`strengthThresh=30` is the first-derivative magnitude threshold; `expectDist=34` tells the scorer the expected width scale of the pin; `edge1Idx=1, edge2Idx=0` means take only the first edge and do not form an edge pair. The 32.73 px width in Section 20.1 comes from these two calls on pin A. One measured caveat: the pin's left-edge gradient is strong and stable only within y [780,920]; outside this range the contrast at the pin's tip/root collapses and the caliper raises 122504007; the ROI and search band must fall within the segment where the edge is clear — this is an extra piece of homework that a real sample imposes relative to an idealized synthetic image.

The `LinePointsLocator` comparison route has triple value: it measures the true width perpendicular to the edge (the SDK caliper can only measure horizontal width, with distortion growing with tilt); it hands back the subpixel point of each search line to the user, making it easy to self-compute per-row scatter and straightness; and it incidentally gives the edge tilt (measured 0.16° for pin A). There is one implementation detail: for near-vertical edges, fit $x = ay + b$ instead, to avoid `FitLine`'s defect on vertical point sets (Chapter 14):

```
SCIMV::SciSvLineLocator loc;
SciPointArray pts;
long rc = loc.LinePointsLocator(src, roi, region, /*strengthThresh*/30,
             /*direction left->right*/2, /*polarity dark->bright*/0, /*edgeWidth*/3,
             /*projectWidth*/1, /*edgeType best*/2, /*searchLineCount*/16,
             /*findPointType first derivative*/1, &pts);
// self-fit x = a*y + b on pts; distance of the two fit lines at the ROI center height / sqrt(
```

It is not uncommon for a commercial SDK's caliper module to carry such quirks; the countermeasure is the same as in Chapter 14: recompute the key quantities yourself from the raw output, and empirically test the usable range parameter by parameter on the real sample. The complete runnable project is in `code/caliper_measurement/`.

Industry Case: Glass Width Measurement at 0.01 mm Tolerance

A cover-glass edge-grinding line requires a width tolerance of ± 10 m at an imaging resolution of 5 m/pixel — the tolerance band converts to ± 2 px, and by the 1/10 rule the repeatability must reach 0.2 px. The single-shot caliper repeatability after projection averaging is about 0.07 px (0.35 m), seemingly with ample margin, yet production data exposed another enemy: the day-night temperature swing of the workshop makes the lens barrel and frame expand and contract, and the measurement mean drifts 0.5 m over a day — five times the repeatability scatter. Good precision does not mean high accuracy: scatter is "how much a hundred measurements today differ", while drift is "how much one measurement in the morning and one in the afternoon differ", and the latter is exactly what the repeatability metric cannot see. The final solution was to place a standard gauge block on the line and re-measure it every two hours, correcting the scale factor in real time — gauge calibration is the bedrock of measurement, the same point as "precision is not accuracy" stressed in Section 20.4 of this chapter.

20.6 Summary

- **The caliper is a directed one-dimensional measuring tool**: ROI + search direction + polarity frame the problem, and projection, differentiation, and parabolic interpolation give the subpixel edge — it shares roots with the search-line machinery and is the one-dimensional specialization for "we know where the edge is and need to measure it accurately". On a real Micro-USB connector this chapter measured adjacent pin widths of 32.73 / 32.24 px, a center pitch of 73.02 px, and a gap of 40.53 px.

- **Subpixel information lives in the gradient**: the lens PSF and photosite integration smear the edge into a gray ramp whose pixel values continuously encode the edge phase; an 8× magnification of the real pin edge reveals a clear transition band, and parabolic interpolation decodes the subpixel position from it.
- **Repeatability is the lifeline of measurement, and multi-search-row averaging is its cheapest amplifier**: the per-row subpixel scatter is 0.37 px, and after fitting a straight line over 20 rows the pin width repeatability drops to 0.073 px ($0.37/\sqrt{20}$), the mechanism being the $\sqrt{N}$ law; composite quantities (pitch, gap), by combining the errors of two independent ROIs, amplify the scatter nearly fivefold to 0.37 px.
- **Precision is not accuracy**: a real sample with no certified ground truth can only give precision (0.07 px), not accuracy; accuracy requires anchoring with an external standard part. The per-row scatter also mixes in the real taper of the stamped pin (about 2 px), which must not be uniformly averaged away as noise.
- **Keep an empirical attitude toward the SDK's caliper module**: this version of `Caliper()` has polarity semantics opposite to the comment, an edge pair that outputs no width, and an always-vertical fit line, so engineering works around it with two single-edge calls; and one must empirically measure the usable segment where the edge is clear on the sample (the pins are measurable only within y [780,920]). The `LinePointsLocator` self-fit route provides perpendicular width and tilt and is a more controllable comparison.

For the accuracy limits of edge localization and the bias-and-variance analysis of various subpixel methods, see the chapters on one-dimensional edge extraction and accuracy evaluation in the work by Steger et al. (Steger, Ulrich, and Wiedemann 2018). One classic origin of subpixel edge localization is Haralick's step-edge detection via the zero crossing of second directional derivatives under a facet model, which systematically characterizes how edge phase is recovered analytically from a local gray-level polynomial (Haralick 1984). As for the chapter's recurring theme that "precision is not accuracy" and the evaluation of re-

peatability and reproducibility (gauge R&R), these belong to gauge capability analysis: AIAG's Measurement Systems Analysis reference manual lays out the standard procedures for assessing repeatability, reproducibility, bias, and linearity, and is the authoritative bridge from this chapter's scatter numbers to production-line judgment specifications (Automotive Industry Action Group 2010).

21 Geometric Measurement

The lines and circles fitted in Chapter 14 are, in the production line's eyes, mere intermediates. What gets signed off on an inspection report is never "the line's direction vector" or "the circle's parameters", but numbers like **47.8 px from the camera hole to the top edge, a hole roundness of 0.25 px, whether the panel's corners are square (a right-angle error of 0.09°)** — numbers that can be compared directly against the drawing's tolerances and decide pass or fail. Getting from primitives to these numbers takes one more stage: **geometric measurement** — applying exact geometric formulas to the fitted points, lines, and circles to obtain distances, angles, and intersections, and then evaluating parallelism, roundness, and straightness according to the definitions of geometric tolerances. The complete measurement chain is: **image → edge points → geometric primitives → geometric quantities → judgment**. This chapter cares about two things: how the error at each stage propagates into the final numbers; and how the "band-width" class of geometric tolerance quantities differs fundamentally, in statistical character, from the "position" class of dimensional quantities.

The object of study (Figure 21.1) is a real **phone cover glass** (a Smart3 "geometric relations" sample image, 2592×1944 single-channel gray): a vertically placed rectangular panel on a white background, whose outer silhouette is a dark bezel, four clean straight edges meeting at the four corners through **genuine rounded fillets**; the top bezel embeds a circular camera hole and an elongated earpiece slot, and the bottom carries a Home-button bonding pad. This is an extremely common consumer-electronics incoming-inspection part — what the line must measure is exactly the hole position, edge clearances, the panel's inner and outer dimensions, the squareness of the four corners, and the roundness of the hole. This chapter runs

the full geometric-formula kit of Section 21.2 on this real part; unlike a synthetic scene, a real part has no analytic ground truth, so everything below is stated as **measured values**, with the part's own geometric self-consistency (the four edges should form an approximate rectangle) serving as a cross-check on measurement correctness.

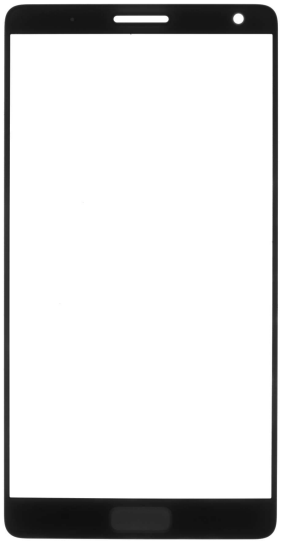


Figure 21.1: The real phone cover-glass sample image (2592×1944): a dark rectangular panel on white, its four straight edges meeting through genuine rounded fillets; the top bezel holds a circular camera hole and an elongated earpiece slot, the bottom a Home-button bonding pad. This chapter measures the geometric relations among its edges, hole, and corner.

21.1 The Measurement Chain and Error Propagation

Lay the measurement chain out stage by stage: at the **edge-point level**, error comes from image noise and the mismatch of the edge model; the **primitive level** does statistical averaging

over many points; the **geometric-quantity level** consists of deterministic closed-form formulas and **introduces no new error of its own** — however much the input primitives are off, the output geometric quantities are off by exactly that much; the **judgment level** compares the geometric quantities against the drawing's tolerances, and the error budget accumulated over the first three stages directly determines the confidence of the judgment (and how much guard band to leave on the tolerance). So when analyzing the accuracy of a vision gauging system, all the work lies in the **error propagation** of the first two stages.

The first stage is the edge points. After projection averaging and parabolic interpolation (the machinery of Chapter 14 and Chapter 20), denote the residual noise of a single edge point along the normal direction by σ_e. In this example an upper bound on it can be read straight from the fit residuals: the all-point RMSE of the top edge's 60 edge points about the `FitLine` least-squares line is only **0.103 px** (the left and right long edges are 0.137 and 0.211 px). This RMSE contains both sensor noise and the slight real form undulation of the straight edge itself, so $\sigma_e \lesssim 0.1$ px is a conservative upper bound — after passing through the whole pipeline, pixel-scale imaging noise has been squeezed below a tenth of a pixel.

The second stage is fitting. Fit a line through N independent points, and the parameter error shrinks as $1/\sqrt{N}$; for the **direction** there is a second lever as well — the span L of the point set. For N points spread uniformly over a span L, the standard deviation of the least-squares line's direction is approximately

$$\sigma_\theta \approx \frac{\sigma_e}{L}\sqrt{\frac{12}{N}},$$

with L in the denominator: the same points, measured over a longer span, give a more accurate angle. Plugging in this example's long edges ($\sigma_e \lesssim 0.1$ px, $N = 90$, $L \approx 1060$ px) yields $\sigma_\theta \approx 0.002^\circ$ for a single line; the angle between two independent lines picks up another factor of $\sqrt{2}$, giving a **measurement noise floor of about** 0.003°. This floor is the point: the measured angle between the left and right long edges is **0.065°**

and between the top and bottom short edges **0.237°** — both far above the 0.003° floor, showing that they are the edges of a **real part** that are not strictly parallel (genuine geometric deviations of the panel's stamping/bonding), not the random jitter of measurement noise. The measurement has resolved the part's true shape, which is exactly what an error budget is for: only once you know where the noise floor lies can you judge whether a reading is real signal or noise.

Primitive extraction itself reuses the pipeline of Chapter 14: the top and bottom short edges each use 60 search lines to extract points followed by a `FitLine` least-squares fit; the left and right long edges each use 90 search lines and are fitted with the x/y-swap workaround (see Section 21.4 for why); the ROIs of all four edges avoid the rounded-corner segments and land only on the straight bodies of the edges. The camera hole is located in one step by `EllipseLocator` along 48 radial search lines, the rim being a bright(inside)→dark(outside) transition of the white hole against the dark bezel.

Pixels and physical quantities: every reading in this chapter is in **pixels (px)**. Converting to millimeters requires one more multiplication by the camera calibration's scale factor (the physical size per pixel); that is the subject of Chapter 5. The sample image carries no calibration, so this chapter evaluates geometric relations and tolerance zone widths only in the pixel domain, fabricating no physical ground truth.

21.2 Distances, Angles, and Intersections

The basic geometric quantities need only three formulas. The **point-to-line distance** from a point $\mathbf{p}$ to the line L through two points $\mathbf{a}$ and $\mathbf{b}$:

$$d(\mathbf{p}, L) = \frac{|(\mathbf{p}-\mathbf{a}) \times (\mathbf{b}-\mathbf{a})|}{\|\mathbf{b}-\mathbf{a}\|};$$

the **included angle** between two lines with direction vectors $\mathbf{u}$ and $\mathbf{v}$ (lines are undirected, so take the acute angle):

$$\theta = \arccos \frac{|\mathbf{u} \cdot \mathbf{v}|}{\|\mathbf{u}\|\,\|\mathbf{v}\|} \in [0^\circ, 90^\circ];$$

the **intersection** is the 2×2 linear system obtained by equating the two lines' parametric equations; a coefficient determinant tending to zero means the two lines are nearly parallel and the intersection is ill-conditioned — which is why engineering interfaces all carry a parallelism-test threshold.

The "corner point" deserves an extra word. The four corners of this panel are **genuine rounded fillets** — there is simply no sharp corner pixel in the image to localize directly; the standard industrial practice is to define the corner as the **virtual intersection** of two fitted straight edges — it need not physically exist on the part, and the theoretical corner of a filleted part is still measurable. Each edge is averaged from dozens of edge points, and the intersection inherits their accuracy in full. This example runs the whole formula kit across the panel: the angle between the top and bottom short edges, the angle between the left and right long edges, the intersection of the top and left edges giving the top-left corner point, the camera hole center's clearances to the top and right edges, the cross-part distance from the corner to the hole center, and the panel's inner/outer width and height obtained from the mean of point-set-to-opposite-edge distances (Figure 21.2). All measured values are in Table 21.1.

Table 21.1: Geometric measurement results on the real cover glass (measured values; the part has no analytic ground truth; units px / °)

Geometric quantity	Measured	Note
Top–bottom edge angle (°)	0.2365	real non-parallelism (0.003° noise floor)
Left–right edge angle (°)	0.0647	real non-parallelism
Top left right-angle error (°)	0.0947	90°−89.9053°, top-left squareness
Top-left corner (px)	(957.475, 121.435)	virtual intersection of two edges
Hole center → top edge distance (px)	47.769	camera top clearance
Hole center → right edge distance (px)	129.780	camera right clearance

Geometric quantity	Measured	Note
Corner → hole center distance (px)	765.908	cross-part point-to-point
Camera hole center (px)	(1721.930, 168.594)	center of 48-point circle fit
Camera hole radius (px)	17.051	fitted circle radius
Panel width L R (px)	893.020	mean of point-set-to-edge distances
Panel height T B (px)	1744.362	mean of point-set-to-edge distances

The credibility of this table comes from **geometric self-consistency**: the pairwise angles measured among the four edges — 0.24° top–bottom, 0.06° left–right, 0.09° top left right-angle error, 0.16° top right — are all within a few tenths of a degree, confirming that the four independently fitted edges do enclose an approximate rectangle; the panel width of 893 px and height of 1744 px also agree with the dark-region silhouette span. The distance quantities are then real incoming dimensions: the camera hole's top clearance of 47.8 px and right clearance of 129.8 px are exactly the critical gaps the camera aperture alignment must hold when such a panel is assembled. All readings stop in the pixel domain — converting to physical dimensions requires multiplying by the camera calibration scale factor (Chapter 5). It bears emphasizing: the geometric-measurement stage is **exact formulas**, and the uncertainty of every number in the table comes entirely from the edge-point noise σ_e propagated through fitting; the formulas themselves are word-perfect.

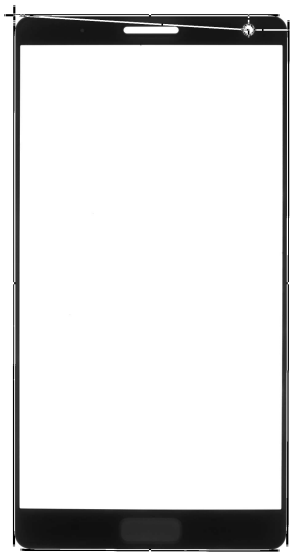


Figure 21.2: Measurement overlay (contrast drawing: black on bright areas, white on dark): the four fitted edge lines extend along the panel's outer silhouette; the camera fitted circle (with center cross) is at top right; the cross at top left is the virtual intersection of the top and left edges (corner point); the short segments drawn from the hole center are the perpendicular distances hole→top edge and hole→right edge, and the long diagonal is the cross-part distance corner→hole center.

21.3 Parallelism, Roundness, and Straightness

Beyond dimensions, drawings carry another class of requirements, written in the frames of geometric dimensioning and tolerancing (GD&T). Their language is not "measure how much" but the **tolerance zone**: the toleranced feature must lie entirely within a region of given width — for **parallelism** the zone is between two lines parallel to the datum line, for **straightness** it is between two parallel lines, and for **roundness** it is the annulus between two concentric circles. The starting point of this language is assembly function: as long as the feature lies wholly within the zone, whatever its specific shape, assembly with the mating part is guaranteed. The universal recipe for evaluating them by vision measurement is one and the same: take the toleranced feature's set of edge points, compute deviations against the evaluation datum, and **max−min is the minimum zone width that contains the point set**. The beauty of a real part is that these zone widths contain **genuine form deviations**, with no need to inject artificial defects.

Parallelism. With the fitted left long-edge line as datum, the max−min of the distances from the right long edge's 90 edge points to the datum is **1.487 px** (over a span of about 1060 px); with the top edge as datum, the bottom edge's parallelism is **2.790 px** (over a span of about 670 px). These two numbers can be read apart: the short edges' parallelism comes almost entirely from that 0.237° real angle — $670 \times \tan(0.237^\circ) \approx 2.77$ px, differing from the measured 2.790 px by only 0.02 px, with the remainder being the bottom edge's own form undulation; of the long edges' 1.487 px, the angle contributes $1060 \times \tan(0.0647^\circ) \approx 1.20$ px, with the remaining ~0.29 px coming from the long edge's own bow. **Parallelism is the sum of systematic tilt and local form**, taken as the max−min extreme, and it absorbs the datum line's fitting error in full as well.

Roundness. Hole diameter and roundness are the most common judgment pair for assembly-class parts: the former determines the fit clearance, the latter whether aperture alignment and load are uniform. In this example, the max−min of the

radial deviations of the camera hole's 48 rim edge points from the fitted circle (radius 17.05 px) is **0.252 px** — a real, tiny out-of-roundness, not a synthetic defect. Figure 21.3 plots the radial deviations magnified 60× on the nominal circle, and one can see the real rim is not perfectly circular: the deviation is dominated by low-order undulation (slight ovalization plus local bumps and dips), with the bottom and lower-left arcs bulging slightly outward. This "low-order dominated" out-of-roundness shape is the typical fingerprint of a stamping/laser hole-cutting process. A reminder about the datum's role: the least-squares circle is the most stable to sampling noise, but its datum is pulled bodily by any local bump or dip, so the out-of-roundness "account" gets spread unevenly across the arcs — if some arc carries clear impact damage, the adjacent intact arcs end up bulging outward relative to the new datum and the arc directly opposite recedes inward (the industry case of Section 21.4 details a judgment dispute caused by exactly this).

Straightness. The max−min of the left long edge's normal deviations is **0.624 px**, and the right long edge's is **0.872 px** (both 90 points, span about 1060 px). The deviation profile (×40) in Figure 21.4 shows the real shape of the two edges: the left edge (top) is fairly straight, with only small high-frequency undulation; the right edge (bottom) carries a gentle bow plus a local bump. Both are the part's real edge form, with no artificial step — the right edge being more bowed also directly explains the "~0.29 px from edge form" portion of the left–right parallelism of 1.487 px discussed above.

ISO roundness evaluation admits several reference circles: the **least-squares circle (LSC)**, the **minimum zone** circle, the minimum circumscribed circle, and the maximum inscribed circle. The minimum zone circle gives the smallest zone width by definition and is the very embodiment of the GD&T semantics; the least-squares circle is the most stable to compute and the least sensitive to sampling noise, but its zone width is systematically larger and its datum pulled by local bumps and dips.

Extreme-value statistics. As more points are sampled, the larger will be the expected range of the noise, and max − min grows slowly with it — exactly the opposite of averaging-class quantities such as centers and angles, which get more accurate with more points. This is why inspection protocols for roundness and straightness must specify the number of sampling points; otherwise numbers from two machines are not comparable.

21.4 SciVision Implementation

The basic geometric quantities are provided by `SCIMV::SciSvGeometryMeasure`, each API corresponding to one formula of Section 21.2:

```
SCIMV::SciSvGeometryMeasure gm;
double angleDeg = 0, d = 0;
rc = gm.IncludedAngle(t1, t2, l1, l2, /*mode angle between lines*/1, &angleDeg);  // top left s
rc = gm.PointToLineDistance(holeCenter, t1, t2, &d);                               // hole->top
SciPoint corner;
rc = gm.LinesIntersection(t1, t2, /*lineType line*/0, l1, l2, 0,
```

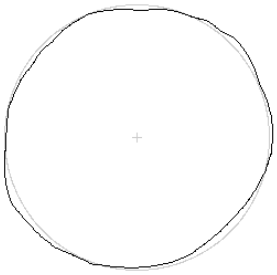


Figure 21.3: Polar roundness deviation plot (radial deviations ×60 superimposed on the nominal circle): gray is the nominal circle, black is the real camera-hole rim (max−min = 0.252 px, 48 points); the out-of-roundness is dominated by low-order undulation, with the bottom and lower-left arcs bulging slightly outward.

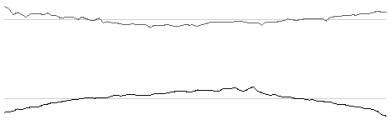


Figure 21.4: Straightness deviation profile (normal deviations ×40, horizontal axis is position along the edge): top, the left long edge (0.624 px, fairly straight); bottom, the right long edge (0.872 px, with a gentle bow and a local bump).

```
                          /*parallelThresh*/7, &corner);                          // virtual co
double dMin = 0, dMax = 0; SciPoint pNear, pFar;
rc = gm.PointsToLineDistance(rightPts, l1, l2, &dMin, &dMax, &pNear, &pFar);
double width = 0.5 * (dMin + dMax);    // panel width (mean)
double parallelism = dMax - dMin;      // left|right parallelism zone width
```

Lines are uniformly represented by two endpoints (a pair of `SciPoint`). With `mode=1`, `IncludedAngle` returns the angle between lines, falling in [0°, 90°] — when the top and left edges are nearly perpendicular it returns about 89.9°, and the squareness is 90° minus that. In `LinesIntersection`, `parallelThresh=7` is the parallelism-test threshold: when the two lines' directions are too close, the intersection is ill-conditioned and the interface simply refuses to output one — in this example the top and left edges are nearly perpendicular, far from that threshold. `PointsToLineDistance` returns in one call the minimum and maximum point-set-to-line distances along with the corresponding points: the mean is the nominal distance to the opposite edge (panel width/height), and max−min is the parallelism zone width. The hole is located in one step by `SciSvEllipseLocator::EllipseLocator`, whose four output point arrays (fit/effective/rejected points) must all be passed as real entities, or rc=120001015.

Two engineering reminders. First, **the `Roundness/Straightness` of `SciSvGDTTools` return normalized [0,1] scores** (closer to 1 means "better"), not GD&T tolerance zone widths: this chapter measured a roundness score of **0.9964** for the camera hole; and the straightness scores of the left and right long edges are **both 1.000000** — even though their real zone widths are plainly 0.624 px and 0.872 px, nearly 40% apart, the normalized score flattens the two into the same saturated value, leaving no way at all to set a threshold against a drawing callout like "straightness 0.02 mm". Nor is there any published conversion formula between score and zone width; they can only serve as a relative trend reference among parts of the same type, and if you must use them for monitoring you first have to calibrate a "score-to-zone-width" curve from reference samples of known zone width. **Zone-width values used for tolerance judgment are always computed as**

max−min from the point arrays yourself (which is where every number in this chapter's text comes from). Second, the output distance array `dstPtPositionArr` of `FitLine` **is not the signed perpendicular distance from each point to the line it returns**; its semantics are opaque, so for straightness you must write your own "signed point-to-line distance" (`devToLine`) and take its range. One old pitfall is worth restating in passing: `FitLine` returns a degenerate result for strictly vertical point sets, so the left and right vertical long edges first swap x/y on all points, fit, and then swap the resulting line's endpoints back. The complete project that generates all of this chapter's images and numbers lives in `code/geometric_measurement/`.

Industry Case: The Roundness Evaluation Dispute

A bearing plant's raceway roundness judgment once reached a deadlock: the customer's incoming inspection used a roundness tester sampling thousands of points densely, evaluated by the minimum zone method; the production line's vision system used a few dozen edge points, evaluated by the least-squares circle. One part, two numbers — the line's value was systematically about 15% lower (sparse sampling misses peaks and valleys and pulls the reading down, an effect that outweighed the opposing effect of the least-squares datum's systematically larger zone width), parts the line passed were rejected by the customer, and the two sides argued for months. The eventual resolution lay not in algorithms but in protocol: the inspection protocol explicitly recorded the evaluation method (minimum zone vs least squares) and the number of sampling points, and a calibrated conversion margin was set for the line's values. The lesson takes one sentence: zone-width quantities like roundness and straightness have no "true value" detached from the evaluation method — **a reported geometric quantity must be reported together with its evaluation method and sampling conditions**.

21.5 Summary

- **The measurement chain is "image → edge points → primitives → geometric quantities → judgment"**: the geometric-measurement stage is exact formulas and introduces no new error; the final accuracy comes entirely from the edge-point noise σ_e propagated through fitting ($1/\sqrt{N}$); on the real panel this chapter measured edge-fit RMSEs of 0.1–0.28 px.
- **Fix the noise floor first, then judge real signal**: the long-edge direction error is $\sigma_\theta \approx (\sigma_e/L)\sqrt{12/N} \approx 0.003°$; the measured left–right non-parallelism of 0.065° and top–bottom of 0.24° are both far above the noise floor, genuine part geometry rather than jitter.
- **Virtual intersection measures the corner of a filleted part**: the panel's four corners are genuine rounded fillets, and the corner point is given by the virtual intersection of the top and left edges at (957.5, 121.4), inheriting the accuracy of each edge averaged over dozens of points.
- **Zone-width quantities (parallelism/roundness/straightness) are extreme-value statistics with real shape**: the left–right parallelism of 1.487 px splits into an angle contribution of 1.20 px plus 0.29 px of edge form; the camera hole's real roundness is 0.252 px and the left/right long edges' real straightness is 0.624/0.872 px — all from the part itself, with no synthetic defects.
- **Recompute key quantities from the raw point arrays yourself**: the SDK's GD&T tools return normalized scores (two edges whose zone widths differ by 40% both score 1.000000), and `FitLine`'s distance array is not the true perpendicular distance — numbers used for tolerance judgment must be computed from the points up.

For a systematic treatment of uncertainty analysis for geometric quantities and subpixel measurement, see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018). The parallelism, roundness, and straightness tolerances used in this chapter have their symbol language and tolerance-zone definitions fixed by geometric dimensioning and tolerancing (GD&T) standards: the international system is ISO 1101, which gives the

rules for indicating and interpreting tolerances of form, orientation, location, and run-out (International Organization for Standardization 2017); the North American system is ASME Y14.5, an equally authoritative statement of the same semantics on engineering drawings (American Society of Mechanical Engineers 2018). The "dispute over the evaluation datum" that Section 21.3 and the industry case keep touching on — the least-squares circle and the minimum zone circle yielding different zone widths — is treated specifically in the form-error fitting literature; Moroni and Petrò compare the principles and costs of various minimum-zone fitting algorithms, a good entry point for connecting this chapter's max-min zone widths to standardized evaluation methods (Moroni and Petrò 2008).

22 Intensity, Color, and Gap Measurement

Not every measurement is about geometry. When an LED backlight panel comes off the line, the question is not "are the LEDs in place" but "are they bright enough, and is the brightness even"; for a batch of injection-molded housings, what must be checked is whether the color deviation from the reference swatch exceeds the limit; for a row of connector pins, the concern is whether adjacent gaps are consistent — any spot that is too wide or too narrow means a risk of poor mating. These three classes of problems fall under **intensity measurement**, **color measurement**, and **pitch measurement** respectively. The objects differ, but the skeleton is exactly the same: **compute statistics inside an ROI → compare against the nominal value → judge by tolerance**. This chapter walks through that pipeline with three actually-run experiments. Figure 22.1 is the scene of the first experiment: four LED spots, one of which is abnormally dim — but by eye alone, can you say for certain which one, and by how much?

22.1 Intensity Measurement

The essence of intensity measurement is **computing statistics over the gray values inside an ROI**. The most common statistics each have their role: the **mean** answers "is it bright overall," and is the first choice for pass/fail judgments on brightness; the **standard deviation** answers "is it uniform inside," useful for spotting local dead pixels or stains; the **minimum/maximum/median** characterize the two tails and the center of the distribution — the median is insensitive to a few outlier pixels and is more robust than the mean when the surface occasionally shows glints or dust. A single measurement

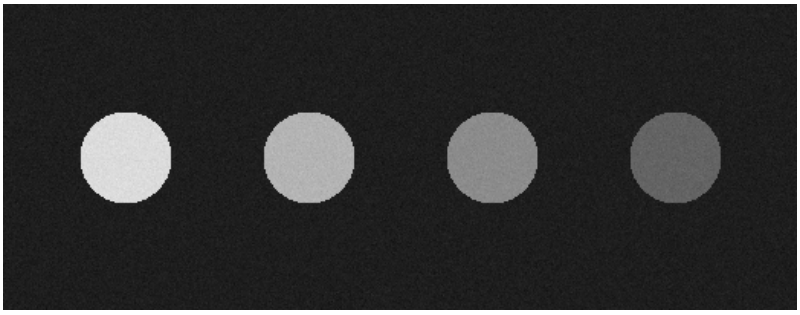


Figure 22.1: Intensity measurement scene: four LED spots with nominal intensities 220, 180, 220, and 100, with Gaussian noise of $\sigma = 4$ superimposed. LED #2 (third from the left) should nominally be 220 but is actually only 140 — a mere 40 gray levels away from LED #1's normal value of 180, hard to call by visual inspection.

call returns this whole set of statistics; take whichever you need.

The experimental scene is shown in Figure 22.1: four LED spots of radius 30 px with nominal intensities {220, 180, 220, 100}, where LED #2 is actually rendered at only 140 (simulating a dim-LED defect), and Gaussian noise of $\sigma = 4$ is added over the whole image. We place a circular ROI of radius 24 px at the center of each LED — deliberately one ring smaller than the spot, to keep the transition edge and the background out of the statistics. The SDK measures means of 220.11, 180.09, 139.97, and 100.06, with a maximum deviation from ground truth of only 0.11 gray levels; the measured standard deviations of 3.95-4.10 agree closely with the injected noise $\sigma = 4$. This shows that as long as the ROI is clean, the precision of gray-value statistics is limited almost solely by the noise itself — averaging over a thousand-plus pixels compresses the per-pixel $\sigma = 4$ random error down to the 0.1 level.

For the judgment step, we adopt the rule "**a measured mean more than 40 gray levels below the nominal value is judged a dim LED**." The four LEDs have different nominal values, so the comparison must be made LED by LED against each one's own nominal — not against a single global thresh-

The standard error of the mean shrinks as $\sigma/\sqrt{N}$: with about 1800 pixels in the ROI, $4/\sqrt{1800} \approx 0.09$, consistent with the measured maximum deviation of 0.11. **Measurement precision can be far better than single-pixel noise** — this is exactly the value of statistical measurement.

old: LED #2's 140, compared against LED #3's nominal 100, would actually look "bright." The run flags exactly one LED — #2 — as DIM DEFECT (Figure 22.2). The tolerance of 40 is not pulled out of thin air: it should be anchored to **the brightness variation the process allows** (for example, the within-bin brightness spread after LED binning plus the drive-current tolerance), with additional margin for the measurement's own uncertainty. Set the tolerance too tight and normal batch-to-batch variation gets falsely rejected; too loose and real defects slip through.

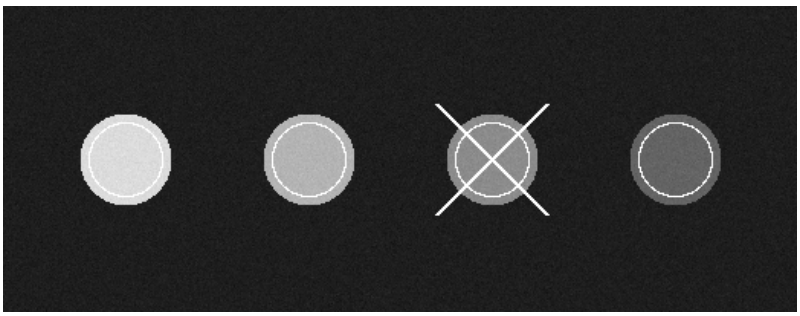


Figure 22.2: Intensity measurement result: the white circles are the circular ROIs of each LED (radius 24 px); LED #2's measured mean of 139.97 is more than the tolerance of 40 below its nominal 220 and is flagged as a dim-LED defect (diagonal cross).

One final reminder: intensity measurement measures "pixel gray value," and pixel gray value = the object's reflectance/emission characteristics × illumination × camera gain and exposure. The algorithm only cares about the first factor; any drift in the latter two passes straight into the measurement result. Illumination aging, voltage fluctuation, and ambient light leakage will all make the same panel measure different brightness at different times (Chapter 4). The robust practice on a production line is to fix an **intensity reference patch** inside the field of view: in every image, first measure the reference patch, then normalize the target reading by its value, canceling the illumination drift.

This experiment uses 8-bit images, with generous gray-level spacing between the 4 LED bins. If pass and fail for the object under test differ by only a few gray levels (e.g., transmittance grading of coated glass), the 256 levels of 8-bit quantization become the precision bottleneck, and 10/12-bit acquisition should be used instead (see Chapter 1).

22.2 Color Measurement

The most direct way to extend intensity measurement to color images is to compute statistics on the R, G, and B channels separately. The experimental scene consists of four 80×80 color patches — blue, green, red, and orange-red — with noise of $\sigma = 5$ added per channel (Figure 22.3). Measuring a 64×64 rectangular ROI inside each patch, the three-channel means deviate from ground truth by at most 0.25, and the standard deviations deviate from $\sigma = 5$ by at most 0.12 — channel-level statistics are just as precise as in the grayscale case.

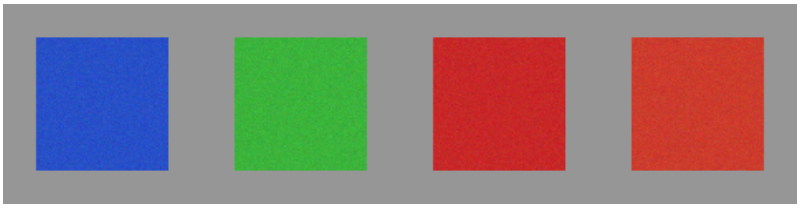


Figure 22.3: Color measurement scene: four patches — blue, green, red, orange-red — with channel noise $\sigma = 5$. Patches 3 and 4 (red and orange-red) look very close to the eye and are the protagonists of this section's color-difference quantification.

But "how much each channel differs" is not the same as "how different it looks to the human eye." RGB space is perceptually non-uniform: the same Euclidean distance of 20 units is almost imperceptible in the green region yet jarring in skin tones. Industry's standard language for color deviation is the **color difference ΔE in CIELAB space**. CIE76 defines it as the Euclidean distance in Lab coordinates:

$$\Delta E_{76} = \sqrt{(\Delta L^*)^2 + (\Delta a^*)^2 + (\Delta b^*)^2},$$

The Lab space is constructed through a nonlinear transform so that equal ΔE corresponds roughly to equal perceived difference. Empirically, $\Delta E \approx 2.3$ is the **just-noticeable difference (JND)** — two colors closer than that cannot be told apart by an ordinary observer even side by side.

In the experiment, the BGR ground-truth values of the red and orange-red patches differ by only (5, 20, 5). Computed from the measured means: the BGR Euclidean distance is 21.02, and after conversion to Lab, **CIE76** $\Delta E = 5.85$ — about 2.5× the JND. This is precisely the typical regime of "close to the eye, separable by instrument": picking these two swatches apart visually is a struggle, while a measurement system distinguishes them reliably and reports a continuous deviation. A note in passing: the SDK does not provide Lab output; the ΔE here is converted in code via sRGB (D65 white point) → XYZ → Lab — see the companion project for the conversion formulas.

In practice there are two SDK conventions you must know. First, with `colorSpace=0`, the mean/standard-deviation arrays returned by `MeasureColor` are ordered **R, G, B** — exactly the reverse of the image's BGR storage order; treating index 0 as B will swap red and blue. Second, in the HSV output of `colorSpace=2`, hue is not the familiar 0-360 degrees or 0-180 (the OpenCV 8-bit convention) but **degrees × 256/360**, with a range of 0-256; pure red's hue wraps around at 0 and 256, and the measured red patch returns 256.0 rather than 0. Hue-threshold judgments must handle this wraparound, or red targets will be missed wholesale.

Division of labor between color measurement and color matching (Chapter 18): measurement answers "by how much it deviates," outputting a continuous quantity, suited to quality grading and process monitoring; matching answers "which class it is," outputting a category label, suited to sorting. The same production line often needs both.

22.3 Gap and Pitch Measurement

The representative third scenario is connectors, pin headers, and IC leads: **large numbers of equally spaced repetitive structures**, where the defect is typically that one of them is tilted, shifted, or missing. Setting up a caliper for each one (Chapter 20) certainly works, but a 64-pin connector would need 64 calipers — tedious and inefficient. The **pitch measurer** is the "batch caliper" designed for this scenario: within a scan band (band ROI), it extracts **all** edges matching the polarity convention along a single direction, pairs them as "rising edge - falling edge," obtains each pin's width and center from each pair, and subtracts adjacent centers to get the **pitch** sequence — measuring the whole row in a single call.

The experimental scene consists of 8 vertical bright pins (width

14 px, nominal center spacing 40 px), where pin #5 is shifted 3 px to the right as a whole to simulate a position defect, with $\sigma = 4$ noise added (Figure 22.4). The scan band spans all the pins, the direction runs left to right, and edge 1 takes black→white polarity (a pin's left edge) while edge 2 takes white→black (the right edge).

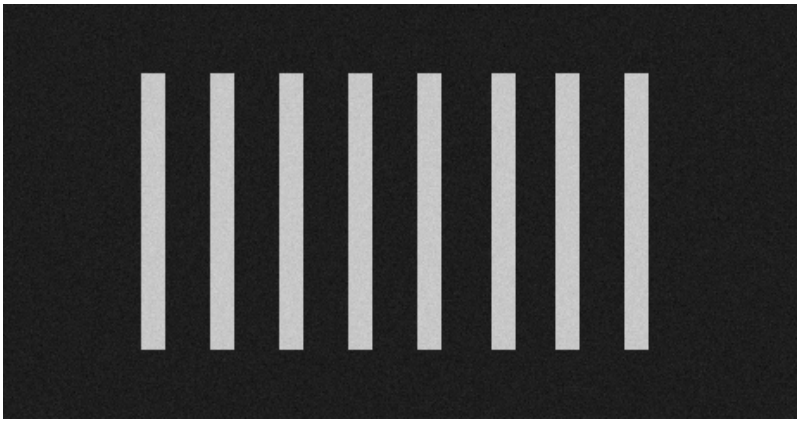


Figure 22.4: Pitch measurement scene: 8 vertical bright pins, width 14 px, nominal pitch 40 px; pin #5 (sixth from the left) is shifted 3 px to the right — nearly invisible to the eye.

Measured results: the 8 pin widths come out at 13.98-14.03 (ground truth 14), with center localization error 0.01 px; the 7 pitches are, in order, 40.00, 40.02, 39.99, 40.01, **43.00**, **37.00**, 40.00. Judged against "nominal 40 ± 1.5 px," exactly the two segments 4-5 and 5-6 are out of bounds, consistent with the SDK's own outputs maxPitchIdx=4 and minPitchIdx=5. The result is shown in Figure 22.5.

Note a pattern here that is worth committing to memory: **one position defect shows up as two pitch anomalies**. Pin #5 shifting 3 px to the right makes the pitch on its left become 43 and the one on its right 37 — one over-large segment immediately followed by an over-small one, with deviations equal in magnitude and opposite in sign. So when tracing a defect location back from the pitch sequence, do not report "segment 4-5 out of tolerance" and "segment 5-6 out of tolerance" as two separate defects; instead, find **the common endpoint of**

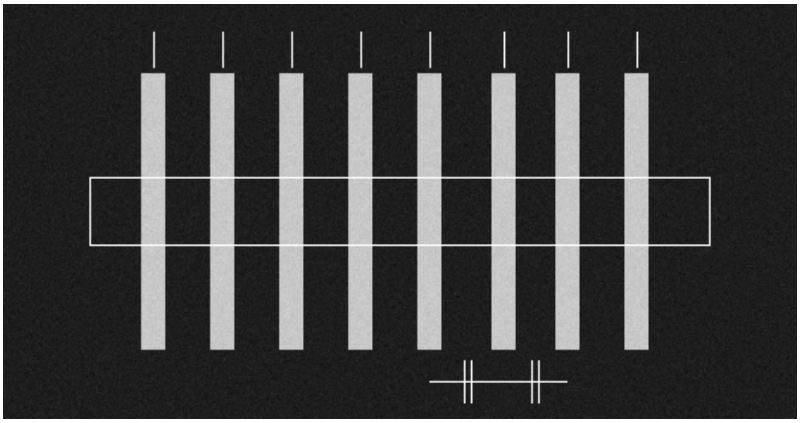


Figure 22.5: Pitch measurement result: the short tick marks at the top are the measured pin centers, the outer frame is the scan band; the two anomalous pitches (segment 4-5 at 43.00 px and segment 5-6 at 37.00 px) are marked with double vertical lines and bottom bars — their common endpoint is the shifted pin #5.

the anomalous pair: pin #5, shared by both anomalous segments, is the true culprit. Conversely, if a single isolated pitch is anomalous while its neighbors are normal, the more likely cause is an abnormal pin width or an edge-extraction error — worth re-checking with a different criterion.

22.4 SciVision Implementation

Pitch measurement shares the same subpixel edge kernel as the caliper — the 0.01 px center error in this experiment is of the same order as the single-edge localization precision in Chapter 20. The difference is only in scheduling: a caliper measures one edge pair at a time; the pitch measurer measures a whole row at once.

The three classes of measurement are handled by `SciSvIntensityMeasurement`, `SciSvColorMeasurement`, and `SciSvPitchMeasurer` respectively. Intensity measurement:

```
SCIMV::SciSvIntensityMeasurement im;
SciROI roi;
SciPoint ctr(LED_CX[k], LED_CY);
roi.GenCircle(ctr, 24.0);                    // circular ROI, radius 24
double avg, sd, mn, mx, med, pixelTotal;
long rc = im.MeasureIntensity(img, roi, /*lower*/0, /*upper*/255,
```

```
                                NULL, &avg, &sd, &mn, &mx, &med,
                                NULL, &pixelTotal, NULL, NULL);
// pixelTotal: SDK output, measured 3324, not the ROI's geometric pixel count ~1810 - semantics
```

`lower/upper` are the gray-level gates for inclusion in the statistics (0..255 means all pixels) and can be used to exclude overexposed points; the outputs are, in order, mean, standard deviation, minimum, maximum, median, and `pixelTotal` (an SDK output, measured at 3324, which is not the ROI's geometric pixel count of ~1810 — its semantics are unclear, so do not use it as an area in engineering); pass `NULL` for any output you do not need.

Color measurement (note that the two calls must use different ROI variables — the parameter is a non-const reference):

```
SCIMV::SciSvColorMeasurement cmeas;
SciROI roi;  roi.GenRect1(tl, br);              // GenRect1's bottom-right corner is an exclus
SciVarArray avgRGB, stdRGB;
long rc = cmeas.MeasureColor(img, roi, /*colorSpace*/0,   // 0=RGB, 2=HSV
                             0, 255, 0, 255, 0, 255,      // per-channel statistics gates
                             /*model*/0, NULL, &avgRGB, &stdRGB, NULL, NULL);
double B = avgRGB[2].D(), G = avgRGB[1].D(), R = avgRGB[0].D();  // return order is R,G,B!
```

Pitch measurement:

```
SCIMV::SciSvPitchMeasurer pm;
SciROI band;  band.GenRect1(SciPoint(50,100), SciPoint(410,140));  // scan band spanning the pi
EdgeDirection dir;   // left to right; edge 1 black->white (left edge), edge 2 white->black (ri
dir.direction1 = 2; dir.direction2 = 2; dir.polarity1 = 0; dir.polarity2 = 1;
EdgeFilter filter;
filter.searchLineCount = 1; filter.edgeWidth = 2; filter.projectWidth = 2;
filter.sensitivity = 30; filter.strengthThresh = 5; filter.strengthLimit = 255;
double avgWidth, avgPitch;  SciVarArray widthArr, intervalArr, pitchArr, ang1, ang2;
SciPointArray edge1, edge2;  int maxW, minW, maxP, minP;
long rc = pm.PitchMeasurer(img, band, region, dir, /*pattern*/0, filter,
                           0, 0, /*widthThresh*/5.0, /*widthLimit*/30.0,
                           /*pairsCount*/8,
                           &avgWidth, &avgPitch, &widthArr, &intervalArr, &pitchArr,
                           &maxW, &minW, &maxP, &minP, &edge1, &ang1, &edge2, &ang2);
```

`widthThresh/widthLimit` bound the legal pin-width interval (5-30 px), filtering out noise edge pairs; `pairsCount` is the expected number of edge pairs; `pitchArr` returns $N-1$ pitches, and `maxP/minP` directly give the indices of the largest and smallest pitch.

The pitfalls hit in practice are summarized below; all are handled in the companion project `code/intensity_color_gap_measurement/`:

- **`MeasureColor` returns in R, G, B order**, the reverse of the image's BGR storage; values must be indexed in reverse;
- **HSV hue is degrees $\times$ 256/360** (0-256), with red wrapping around to 256.0; hue-interval judgments need wraparound handling;
- **`PitchMeasurer`'s edge-point arrays output 2 endpoints per edge** (one each at the scan band's top and bottom rims, interleaved): for 8 pins, `edge1` has length 16 rather than 8; when taking per-pin centers, first average the two endpoints of the same edge.

Industry Case: Backlight Luminance Uniformity

A backlight-module production line performed acceptance on **uniformity** with the 9-point method: 9 measurement points are taken across the screen, and the ratio of the dimmest point's mean to the brightest point's mean must be 80%. Single-point repeatability had always been good, yet customer complaints rose after mass production. Investigation found that the fixture's metal clamping rim reflected light back onto the screen's four corners, systematically inflating the means at the 4 corner points by about 5% — and the corners are precisely where a backlight is dimmest, so the inflated readings pushed the min/max ratio artificially high, and low-uniformity modules that should have been rejected were released. The fix came in two steps: add a light shield to the fixture to eliminate the reflection; and at the same time change each point's statistic from the mean to the **median**, suppressing the pull of any residual glint on the readings. After the fix, the judgments once again agreed with visual evaluation. The lesson: **the choice of statistic is part of the measurement design** — the mean treats every pixel equally, while the median naturally

discards a few outliers; when the environment at the measurement points cannot be controlled, the latter is often closer to "the brightness a person sees."

22.5 Summary

- **Three classes of non-geometric measurement share one skeleton**: statistics inside an ROI → comparison against the nominal value → judgment by tolerance. The ROI must avoid edges and background; the tolerance must be anchored to process variation, not gut feeling.
- **Statistics push precision below the noise**: the standard error of the mean shrinks as $\sigma/\sqrt{N}$ — in this chapter, the intensity-mean error is 0.11, the channel-mean error 0.25, and the pin-center error 0.01 px, all far below single-pixel noise.
- **Color differences must be computed in a perceptually uniform space**: RGB distance does not reflect visual difference; the industrial criterion is CIELAB ΔE, with JND 2.3; red vs. orange-red measured $\Delta E = 5.85$ — close to the eye, separable by instrument.
- **One position defect = two pitch anomalies**: an adjacent anomalous pair in the pitch sequence, equal in magnitude and opposite in sign, points to its common endpoint; the pitch measurer is in essence a batched caliper.
- **Remember three SDK conventions**: color means return in R,G,B order, HSV hue is on the 256 scale and wraps around at red, and pitch measurement's edge points output two endpoints (top and bottom) per edge.

For the radiometric and colorimetric foundations behind intensity and color measurement, and a systematic discussion of edge-pair measurement, see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018). The concepts of CIELAB, white point, and ΔE used in this chapter trace to the authoritative colorimetry classic by Wyszecki and Stiles, where the definitions, data, and derivations of nearly every color space and color-difference formula are collected (Wyszecki and Stiles 2000). For brevity this chapter uses the CIE76 ΔE, whereas

the shop floor more often uses the perceptually more uniform CIEDE2000; Sharma et al. give the complete implementation details, supplementary test data, and several mathematical observations for that formula, a practical reference for upgrading this section's color-difference computation to the current standard formula (Sharma, Wu, and Dalal 2005).

Part VI

Inspection

This part covers algorithms for deciding whether a product passes or fails: blob analysis, contour analysis, shape feature description, and a dedicated treatment of defect detection for real production lines.

23 Blob Analysis

Thresholding (Chapter 7) turned the image into black and white, but the question the production line asks is never "which pixels are white?" — it is: how **many** chip components are on this tray? Which of them are touching each other and need to be re-fed for rework? How big is the largest blob, and where is it? To answer such questions, we must first organize "a heap of white pixels" into "individual objects" — this is the first stop after segmentation. **Blob analysis = connected component extraction + feature computation + filtering and classification**; together these three steps form the backbone of counting, sorting, foreign-object detection, and a good half of the inspection world.

Throughout this chapter, one real industrial sample drives all the experiments (Figure 23.1): a frame from Smart3's "blob-analysis example recipe" — a batch of surface-mount chip components scattered over a dark felt tray, their metallized end-faces glinting bright under coaxial light. The original image is 3840×2748 single-channel grayscale, box-downsampled ×4 to 960×687 to fit the book figures. Roughly 24 components lie in the field at assorted angles, including **one pair touching end-to-end** and **one component clipped by the bottom field edge**; the felt tray, meanwhile, raises a sheet of fine specks under a bright threshold. Our task is to count the components and pick out the touching pair — and every pitfall we hit along the way is a regular visitor on real production lines.

23.1 Connected Components and Labeling

What we call an "object" in a binary image is mathematically a **connected component**: a set of foreground pixels in which

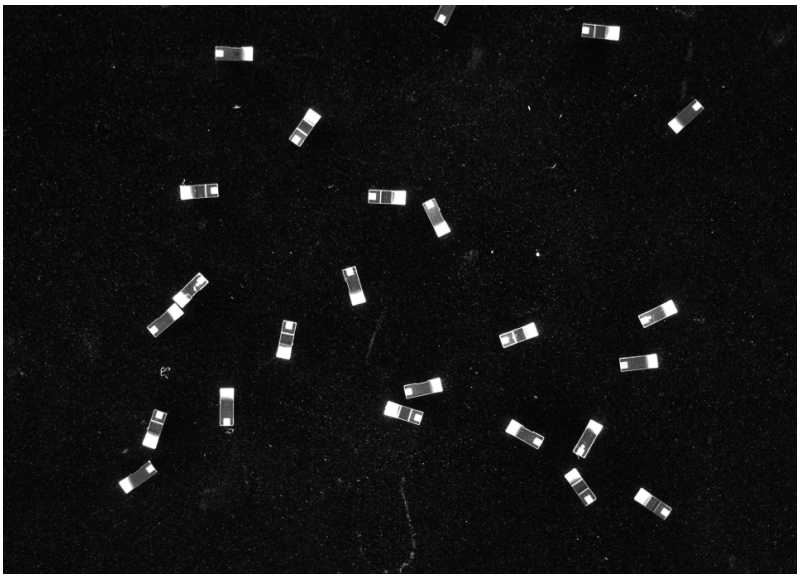


Figure 23.1: Real industrial sample (chip-component bulk inspection, 960×687, downsampled ×4 from 3840×2748): chip components scattered on a dark tray, their metallized end-faces bright. The frame contains one end-to-end touching pair, one component clipped by the bottom edge, and abundant tray-felt specks.

any two can reach each other through a chain of adjacent foreground pixels. The crux lies in the definition of "adjacent" — **4-connectivity** recognizes only the four neighbors above, below, left, and right, while **8-connectivity** also counts the four diagonal neighbors.

This seemingly trivial choice has very real consequences. Two patches that touch only at a single diagonal corner are **one** object under 8-connectivity but **two** under 4-connectivity; a one-pixel-wide diagonal line is a single intact line under 8-connectivity but shatters into a string of isolated points under 4-connectivity. In counting, choosing the wrong connectivity makes the count systematically too high or too low. Industrial libraries almost universally default to 8-connectivity, because the slanted boundaries of real targets, once sampled, naturally connect along the diagonals.

The process of finding each connected component and assigning it a number is called **connected component labeling**. The classic approach is a two-pass scan: the first pass walks through the rows assigning provisional labels to foreground pixels while recording equivalences ("these labels are actually connected"); the second pass merges the equivalent labels into final numbers. Implementations based on run-length encoding are more efficient — each row's consecutive foreground segments are first compressed into runs, then connectivity is established between runs on adjacent rows, so the complexity depends only on the number of runs, not on the area. SciVision's `SciRegion` is exactly such a run-length-encoded region representation: thresholding outputs a `SciRegion` directly, and blob analysis splits it into a `SciRegionArray` — each element of the array is one blob. Figure 23.2 shows this chapter's sample under a **bright threshold** $T = 80$ (pixels of gray level ≥ 80 become foreground): the components stand out as solid blocks, but the dark felt raises hundreds of specks of 1–a-few pixels at this threshold. This image — noise and all — is the input to everything that follows.

Diagonal contact counts as **one** blob under 8-connectivity. The touching pair later in this chapter is a different matter, though — those two components **physically overlap end-to-end**, their pixel sets genuinely merge, and they coalesce under both 4- and 8-connectivity. Another classic convention: when the foreground uses 8-connectivity, the background should use 4-connectivity (and vice versa); otherwise the inside and outside of a closed 8-connected curve would themselves be 8-connected — a topological self-contradiction.

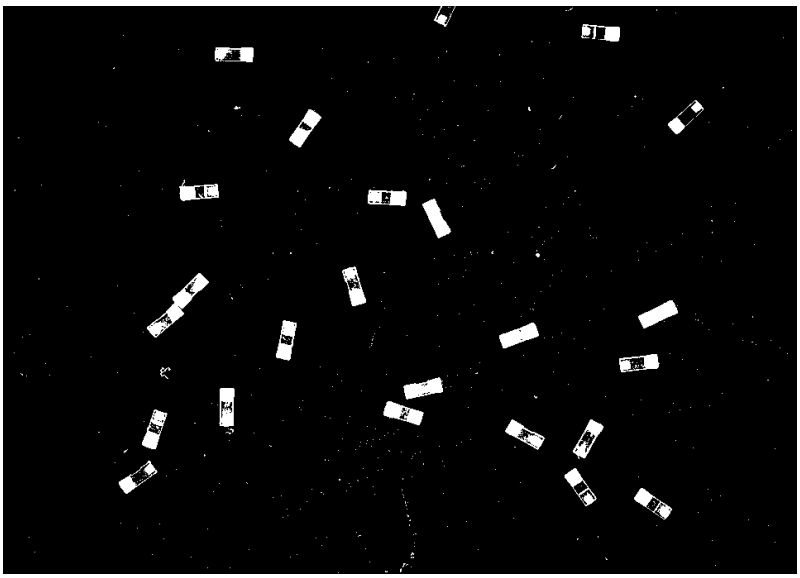


Figure 23.2: Binary image from a bright threshold ($T = 80$, foreground = bright components): the components are clean blocks, but the tray felt simultaneously raises a sheet of specks — those will be removed by the subsequent area filter.

23.2 Blob Features

Once we have the blob array, each blob is still just "a clump of pixels." To classify, we must first measure it into numbers — these are the **region features**. The commonly used features fall into three families.

Size and position. **Area** is simply the pixel count — the cheapest and most robust feature of all; the **centroid** is the mean of the coordinates:

$$\bar{x} = \frac{1}{A} \sum_{(x,y)\in R} x, \qquad \bar{y} = \frac{1}{A} \sum_{(x,y)\in R} y;$$

the **bounding box** gives the smallest enclosing rectangle, commonly used for visualization and rough size gating.

Equivalent ellipse. From the region's second-order central moments one can compute an ellipse with the same moments of inertia as the region; its semi-major axis R_A, semi-minor axis R_B, and orientation angle φ summarize the region's "elongated posture." The ratio of the two semi-axes defines the **anisometry**:

$$\text{anisometry} = \frac{R_A}{R_B} \;\geq\; 1,$$

and it is independent of the target's position and rotation. This chapter's measurements are the best footnote to that claim: the 21 single components, whatever angle they are turned to, all cluster their anisometry in the narrow band **2.7**–**3.4** (a length-to-width ratio of about 2.5 1); whereas the end-to-end touching pair shoots up to **5.26** — nearly twice the nominal value, and the key discriminator we use to flag it later.

Dimensionless shape measures. **Circularity** measures how close a region is to a circle; the common definition is

$$c = \frac{4\pi A}{P^2} \in (0, 1],$$

where P is the perimeter; a perfect circle scores 1, and the more elongated the region or the rougher its boundary, the smaller the value. **Rectangularity** is the ratio of the region's area to that of its enclosing rectangle — an ideal rectangle approaches 1. SciVision's `SciFeatureType` enumeration covers this family quite thoroughly: area, centroid, bounding box, equivalent ellipse $R_A/R_B/\varphi$, anisometry, circularity, roundness, rectangularity, all the way to Hu moments and hole count — this chapter's experiments use 10 of them.

With so many features, which to choose? Three principles: **interpretable** — when a misclassification occurs, you must be able to explain "why" to the process engineer; **sensitive to the differences between targets** — the touching pair and a single component differ in anisometry by nearly a factor of two, which is what makes it worth using; **insensitive to noise and pose** — area and anisometry are both translation- and rotation-invariant, whereas the bounding-box width/height and the axis-aligned rectangularity all change as the target rotates, so they should not be used directly as classification criteria.

One measured lesson from this chapter: SciVision's rectangularity (`SCI_REGION_RECTANGLEDEGREES`) is computed against the **axis-aligned** bounding rectangle, and is therefore **pose-dependent**. For the same batch of components, the squarely-aligned ones score as high as 0.9, while a chip tilted 45° drops to 0.38 (its axis-aligned box is nearly square). So rectangularity here **cannot** serve as a pose-invariant criterion — area and anisometry are the ones that can. Beware too any feature whose denominator contains the **perimeter** (circularity): in a digital image the perimeter is counted along pixel boundaries and is systematically overestimated for small targets.

23.3 Filtering and Classification Experiment

Now let us run the pipeline end to end. Extracting connected components from Figure 23.2 with no restrictions whatsoever yields **479** blobs — the vast majority being tray-felt specks. The first gate is **area filtering**: keep only blobs with area ≥ 150, and the sheet of specks is promptly evicted, leaving **23** component-sized connected components. The second gate is `ignoreROIBoundary` — **discarding blobs that touch the ROI border** — which removes the component clipped by the bottom field edge, leaving **22** to enter classification.

Why must border-touching objects be discarded? Because they are **truncated by the field of view**: only half is imaged, and their area and anisometry are all distorted — classifying on untrustworthy features necessarily produces untrustworthy results. The correct move is not to "try harder to classify" but to admit that this frame cannot see the whole object, and leave it to the next frame or to an adjacent camera. This is precisely

the engineering value of the `ignoreROIBoundary` parameter: natively supported by the SDK, a single boolean shuts this entire class of systematic errors out the door.

The classification rules use only two **pose-invariant** features:

- area $\in [300, 1000]$ **and** anisometry $\leq 4.0 \rightarrow$ **good single component**;
- otherwise $\rightarrow$ **reject** — a touching pair, a foreign object, or a truncated fragment.

The reason we set both an area cap and an anisometry cap is that touching inflates both quantities at once: an end-to-end merge has roughly twice the area of a single component and also roughly twice the anisometry, so either criterion catches it (mutually redundant — and thus more robust). The results are as follows:

Table 23.1: Filtering and classification results (22 blobs; manual check: about 24 components lie in the field, including 1 border-clipped and 1 touching pair)

Class	Decision rule	Detected	Notes
good single	area [300,1000] and aniso 4	21	measured anisometry 2.72–3.41
reject	otherwise	1	end-to-end touching pair: area/anisometry both 2× nominal

Figure 23.3 paints the results back onto the original image: good singles get thin white boxes, and the anomalous touching pair gets a **thick white box with a diagonal cross**. Reading this figure carefully reveals three things: the scattered felt specks have **no box at all** — the work of the area filter; the component clipped at the bottom edge is likewise unboxed — `ignoreROIBoundary` discarded it before classification; and that

crossed thick box at the middle left is the **single anomaly** among the 21 good components — a pair of components touching end-to-end, and the protagonist of the next section.

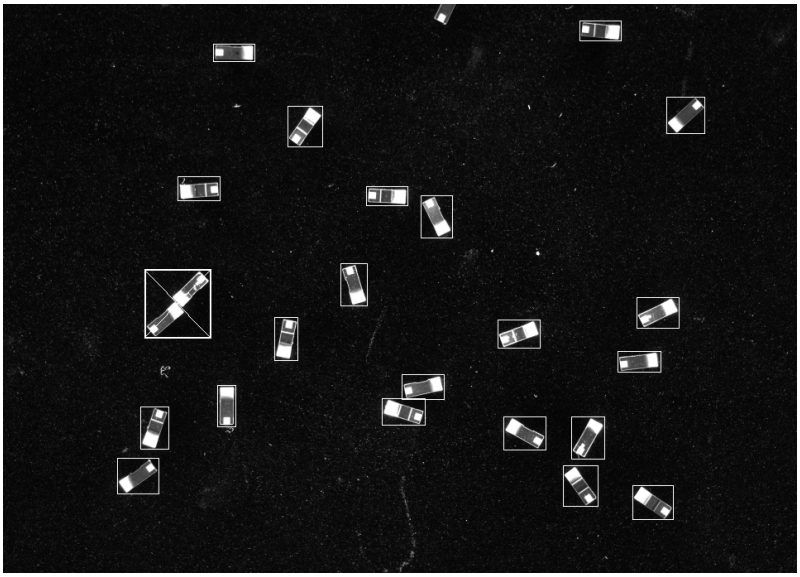


Figure 23.3: Classification overlay: thin white box = good single component (21 of them), thick white box with a diagonal cross = the anomalous touching pair. The felt specks and the bottom-clipped component received no boxes (already filtered out).

23.4 Touching and Merging

That touching pair is a nail already planted in the real sample: two chip components overlap end-to-end, their metallized faces connected at the seam, and after binarization they inevitably merge into one blob. Figure 23.4a shows that area of the binary image magnified 8×: a "bent bar" with a narrow waist in the middle. Connected component labeling is helpless here; what it sees is one object.

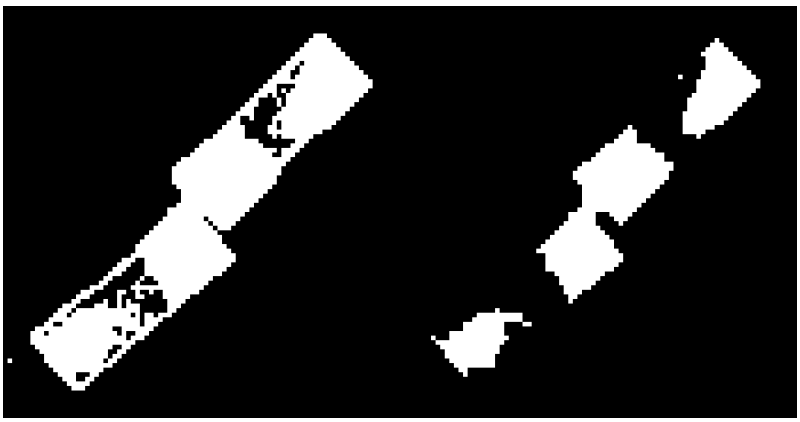
The merged blob's features expose the failure mechanism completely — and hand us the key to detecting it: area **1342 px** (the median area of a good single component is **637 px**, a

ratio of 2.1) and anisometry **5.26** (a single is about 2.8, a ratio of nearly 2). Two pose-invariant criteria raise the alarm at once — which is exactly why it was classified as a reject in the previous section. **Recovering the count**, meanwhile, has a path lighter than morphology: the **area quotient**. Divide the merged blob's area by the nominal single-component area, $1342/637 = 2.11$, round to **2** — and this pair is restored in full. The component count for the whole frame is then 21 (good) + 2 (area-quotient recovery) = **23** (plus the one clipped by the bottom edge, left for the next frame).

The other classic countermeasure is **separation by erosion** (for the principle of erosion see Chapter 8): the merge usually happens at a narrow "neck," and a few erosions should pinch it off. The experiment does exactly that — and teaches a lesson about real structured parts. These chips are not solid bright blocks but structured parts of "**two bright end-faces + a dimmer body**"; a single 3×3 erosion, while pinching off the touching neck, simultaneously severs each single component's end-face from its body. Figure 23.4b is the same area after one erosion: the bent bar did not split cleanly into two — it crumbled into several pieces. And the price lands on the table immediately: the count of component-sized blobs across the whole image jumps from **22 to 32** — erosion **corrupts** the count rather than fixing it.

Touching is not the fault of the connectivity choice — it is an inevitability of feeding: when parts are shaken loose onto the tray, some will always land end-to-end. Worth stressing is the contrast with the diagonal contact of the first section: these two components have **genuinely overlapping pixel sets**, so they merge under both 4- and 8-connectivity; switching connectivity cannot save you here — you can only detect it at the feature level, or break it up at the mechanical level by re-feeding.

The engineering conclusion must be stated bluntly: **on real structured parts, separation by erosion cannot be applied blindly.** In a synthetic scene tangent circles are solid disks, and erosion merely shaves boundaries and splits them cleanly; but the moment a target has internal light/dark structure (metallized face vs body, plating vs base), erosion tears the good part open along its internal "dark seam," distorting count and area together. The robust route on a real line is therefore to walk on two legs: **detect the merge with pose-invariant features** like area and anisometry (flag it as a reject and raise an alarm), and **recover the count with the area quotient rather than the connected-component count**. If you genuinely must cut the merge apart at the pixel level, watershed segmentation on the distance transform is far safer than blind erosion — it cuts along the "ridge" of the distance



(a) Merged: binary image at 8× (b) After one 3×3 erosion

Figure 23.4: The end-to-end touching pair magnified 8×. (a) After binarization the two components join at the seam into a single bent bar with a narrow waist (area 1342 px, anisometry 5.26), merged into one connected component; (b) one 3×3 erosion does pinch off the touching neck, but it also severs each component's bright end-faces from its dimmer body, crumbling it into multiple pieces — the whole-image blob count jumps from 22 to 32, the count corrupted rather than corrected.

map without shaving off area, at the cost of a much heavier implementation and tuning.

23.5 SciVision Implementation

This chapter's pipeline is the collaboration of three classes: `SciSvThreshold` produces the region, `SciSvBlobAnalysis` splits and filters, and `SciSvRegionFeature` computes the feature table.

```
// 1) Bright threshold: keep bright components in [80,255]; dstRegion and binary out together
SCIMV::SciSvThreshold th;
SciImage binImg;  SciRegion fgRegion;
th.ManualThreshold(img, roi, /*lower*/80, /*upper*/255,
                   SCI_THRESHOLD_TYPE_WHITE, 0, &binImg, &fgRegion);

// 2) Blob extraction: filter specks by area at extraction time + discard border-touching obje
SCIMV::SciSvBlobAnalysis ba;
SciRegion mask;                          // default-constructed = mask nothing
SciVarArray minV, maxV, ft;              // the triple for feature-interval filtering
ft.Append(SciVar((int)SCI_REGION_AREA));
minV.Append(SciVar(150.0));  maxV.Append(SciVar(5000.0));
SciVarArray fhF, fhMin, fhMax;           // left empty = no hole filling
SciRegionArray blobs;  SciMatrix featM;
ba.BlobAnalysis(fgRegion, img, roi, mask, minV, maxV, ft,
                /*fillHole=*/false, fhF, fhMin, fhMax,
                /*ignoreROIBoundary=*/true, false, &blobs, &featM);

// 2') Alternatively, extract wide open first, then post-filter by area with BlobFilter
ba.BlobFilter(blobsAll, featAll, SCI_REGION_AREA, 150.0, 5000.0,
              &featArea, &blobsArea);

// 3) Compute features over the blob array (area, centroid, bbox, anisometry, rectangularity..
SCIMV::SciSvRegionFeature rf;
SciVarArray feats;                       // Append the SciFeatureType codes in order
SciMatrix m;
rf.CalRegionFeature(blobs, feats, &m);
```

Several parameters deserve individual comment. `ManualThreshold` takes the gray band `[80,255]`, and `SCI_THRESHOLD_TYPE_WHITE`

means "in-band is foreground" — just right for bright targets on a dark background; its last parameter `dstRegion` is an engineering treat: the segmentation result, as a `SciRegion`, feeds **directly** into the first parameter of `BlobAnalysis`, with no detour through a binary image and re-extraction. The three arrays `minValue/maxValue/featureType` of `BlobAnalysis` constitute the feature-interval filter applied at extraction time (this example sets only an area floor to strip specks); `ignoreROIBoundary = true` is the border-discard described in Section 23.3. `BlobFilter` provides after-the-fact filtering instead — grab the full set wide open first, then tighten as needed, which makes it easy during tuning to see exactly whom each gate filtered out (the numbers 479 → 23 → 22 were printed exactly this way). Note the full-image ROI uses `GenRect1((0,0),(W,H))`: the lower-right corner is **exclusive**, so passing (W, H) covers the whole image, whereas $(W-1, H-1)$ would miss the outermost row and column.

Three pitfalls need to go on record. First, the row/column orientation of the `SciMatrix` output by `CalRegionFeature` is **not documented** — this chapter's companion code auto-detects the orientation by checking which dimension's length equals the number of features before reading values, and you must keep this layer of defense when porting. Second, the rectangularity `SCI_REGION_RECTANGLEDEGREES` is computed against the axis-aligned bounding rectangle and **varies with the target's angle** (measured 0.38–0.93 across same-model components) — it cannot serve as a pose-invariant criterion; for a pose-invariant shape criterion use anisometry. Third, several headers including `SciSvBlobAnalysis` unconditionally define `DLL_EXPORTS` (i.e., they declare with dllexport); placing their `#include` after the dllimport-style headers links fine — verified in practice with no ill effect.

Industry Case: Rework for Touching Parts in Bulk Feeding

An SMD incoming-inspection line used a vibratory bowl to shake chip components onto an inspection tray, then ran blob analysis to count each tray and verify the specification. When the density rose, occasionally two components landed touching end-to-end, merged into one blob after binarization, and the count came up short by 1 — whereupon the system falsely

alarmed "missing part" and stopped the line, even though not a single part was missing. Investigation found that every false-alarm frame contained one "big blob" with roughly twice the area of a normal component and nearly twice the anisometry — the touching pair. The line first tried to cut it apart with erosion, only to find these "end-face + body" structured parts crumbled under erosion and made the count worse. The final fix used no morphology but walked on two legs: a **dual threshold on area and anisometry** flagged the touching pair as anomalous and triggered the bowl to **re-shake the tray** (resolving the merge mechanically), while the "missing-part" criterion was changed from connected-component count to **total foreground area divided by the nominal single-part area, rounded** — the touching pair's area quotient is $1342/637 \approx 2$, recovering exactly the missing part. After deployment the false alarms dropped to zero. The lesson: **in counting tasks, the "area quotient" is more robust than the "connected-component count"** — merging changes the topology but barely changes the area; and on real structured parts, separation by erosion must be used with great caution.

23.6 Summary

- **Blob analysis = connected components + features + filtering** — the standard three-act play that turns a "heap of white pixels" into "countable objects"; the connectivity choice (4/8) decides whether diagonal contact counts as one or two, but a **physical overlap** merges the pixel sets and no connectivity choice can save it.
- **Three principles of feature selection**: interpretable, sensitive to target differences, insensitive to noise and pose. Area and anisometry are the most robust first-line features (this chapter's singles hold anisometry at 2.7–3.4, the touching pair shoots to 5.26); axis-aligned rectangularity and bounding-box dimensions drift with rotation, and perimeter-based circularity is unreliable for small targets.
- **Filtering is the prerequisite for classification**: the area floor removes the felt specks, and

`ignoreROIBoundary` discards border objects that are truncated by the field of view and carry untrustworthy features — each step of this chapter's 479 → 23 → 22 does its own job.

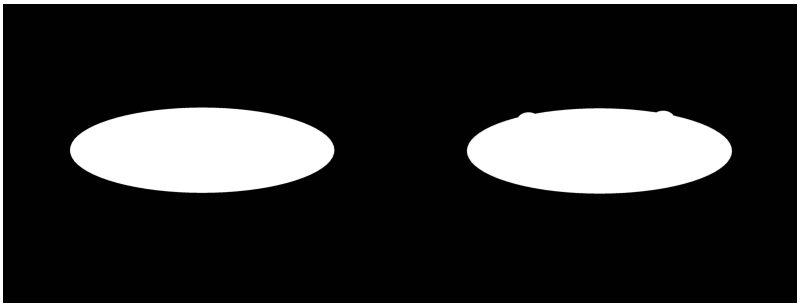
- **Touching is an inevitability of feeding, and separation by erosion depends on the target**: synthetic solid targets split cleanly under erosion, but real "end-face + body" structured parts crumble (whole-image blobs 22→32), the count corrupted rather than fixed. The robust practice is to detect the merge with area/anisometry and recover the count with the area quotient, or switch to watershed; at the mechanical level, re-feed the tray.
- Blob features look only at the pixels' "occupancy," not at the boundary's "shape trajectory" — when classification needs finer contour information, proceed to contour analysis (Chapter 24).

The algorithmic roots of connected component labeling trace back to Rosenfeld and Pfaltz's foundational paper on sequential operations in digital pictures (Rosenfeld and Pfaltz 1966) — connected component labeling, the distance transform, and an early form of the skeleton all appear there; for the topological analysis and run-style extraction of region boundaries, see Suzuki and Abe's border-following algorithm (Suzuki and Abe 1985). A textbook survey of connected component labeling and region features is given by Gonzalez and Woods (Gonzalez and Woods 2018); for a more systematic engineering treatment of region features and morphological segmentation (including watershed), see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

24 Contour Analysis

Blob analysis in Chapter 23 treats a target as a clump of pixels to be tallied — area, centroid, inertia axes — answering questions about "this lump of stuff". But many industrial problems do not ask about the lump; they ask about the **boundary**: how far does this stamped part's outline deviate from the drawing? Are there burrs, bumps, or missing material along the edge? The representation needed here is the **contour** — an **ordered string of subpixel points** laid out one after another along the target's boundary. Ordered means you can walk the boundary point by point, segment it, and localize defect positions; subpixel means accuracy is not nailed down by the pixel grid. The contour is the high-resolution language of defect detection and shape measurement: the region tells you "how much" of a target there is, the contour tells you "what it looks like".

This chapter runs all of its experiments on a set of real industrial sample images (Figure 24.1): an ellipse-shaped workpiece imaged under backlight, yielding a 2592×1944 high-contrast silhouette — a bright white part against a pure-black background, with a narrow, steep gray transition band at the boundary, exactly the ideal input for subpixel contour extraction. There are three images: `001` is the **standard part (the master)**, whose outline is a clean ellipse; `003` and `002` are two **test parts**, each carrying an edge defect — the top edge of `003` has two outward protrusions (excess-material defects), and the bottom edge of `002` has a single larger bump. We will extract all three parts' contours, measure the master's features, perform smoothing/resampling/segmentation operations on the master, and finally compare the test parts against the master point by point — letting the edge defects surface on their own.



(a) The standard part (master, `001`)

(b) The test part (top-edge defects, `003`)

Figure 24.1: Real industrial sample images: a backlit ellipse-shaped workpiece silhouette (2592×1944 grayscale). (a) The standard part: a clean ellipse; (b) the test part: two outward protrusions on the top edge (one on each side, requiring a close look to spot). The backlight renders the part as a high-contrast silhouette with a narrow, steep transition band at the boundary.

24.1 Contour Extraction

Contour extraction stands on the shoulders of edge detection (Chapter 13): edge detection outputs an image of "which pixels are edges", with no order among the pixels; contour extraction then does two more things — **linking**, stringing adjacent edge pixels into chains, and **subpixel refinement**, refining each point's position along the chain from integer to subpixel (the same idea as the subpixel section of Chapter 13: parabolic interpolation over three magnitude samples along the gradient direction; the Steger-style curve extractor introduced there outputs subpixel line contours directly). The product of linking is an ordered point list, and the contour of a closed target joins head to tail into a loop.

The experiment extracts on the full-image ROI with subpixel Canny (hysteresis thresholds 20/40, smoothing coefficient 2.0). The key parameter is the **minimum contour length** `minLen=600`: the black background outside the silhouette has almost no gradient and produces virtually no stray contours,

so the length filter is a safeguard — in the end each of the three parts retains just **1** closed contour, all with **3627 points**: under the high-contrast silhouette, the boundary chain lengths of the three images are nearly identical, with master perimeter **3826.3**, top-defect part **3844.5**, bottom-defect part **3846.0** — the test parts' perimeters running slightly longer than the master is direct evidence that the protrusion defects make the chain take a small detour. In general the number of contour points depends on the chain's path and the noise realization, so **two extractions of the same shape need not, and need not be made to, have the same point count** — the comparison algorithm that follows works by "nearest distance" and does not require the two contours' points to be aligned.

Figure 24.2 overlays the two contours back onto the original images: green for the master, orange for the test part. The subpixel point list hugs the centerline of the gray transition band, and the high-curvature ends of the ellipse are smooth and continuous — evidence that the backlit silhouette faithfully hands the crisp boundary to the subpixel extractor.

A region and a contour are two representations of the same target and convert into each other: take the boundary of a region to get a contour, fill a closed contour to get a region. But the "accessibility" of information is entirely different — connectivity and hole count are best computed on the region, while local boundary deviation and piecewise geometry are best computed on the contour. Choosing the representation is choosing the language of the problem.

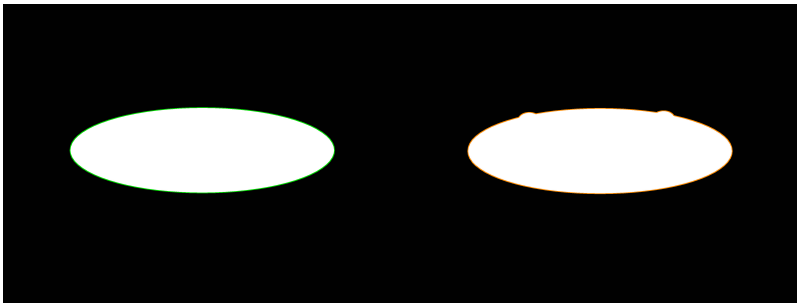


Figure 24.2: The extracted contours of the two parts. Left: the master (green, 3627 points); right: the top-defect part (orange, 3627 points). The backlit silhouette produces almost no stray contours, leaving one closed contour in each image.

24.2 Contour Features

A closed contour carries a full set of geometric quantities corresponding to the Blob features. **Perimeter** is the sum of

adjacent point distances; **area** need not fill pixels and count them — Green's theorem turns the region integral into a loop integral on the boundary (i.e. the shoelace formula), and the ordered point list gives the area directly. The **center** is the geometric midpoint of the bounding box, the **gravity (centroid)** is the centroid of the enclosed region, and the **orientation** comes from the principal axis of the second moments. Two dimensionless shape factors: **circularity** measures closeness to a circle (1 for a circle, smaller as the shape grows more elongated); **rectangularity** is the ratio of area to the smallest enclosing rectangle's area. Measured values for the master contour:

Table 24.1: Measured features of the master contour (`SciSvContourFeature`)

Feature	Measured value
Area	751878.4 px^2
Perimeter	3826.33 px
Center	(1300.53, 950.50)
Gravity	(1300.53, 950.52)
Orientation	$-0.00°$
Circularity	0.321
Rectangularity	0.785
Smallest enclosing rect	1726.7 × 555.0 @ $-89.99°$

This table is itself a self-check, and every number says "this is a standard ellipse". The smallest enclosing rectangle is **1726.7 × 555.0** at orientation $-89.99°$ — a major axis of 1726.7 px horizontal, a minor axis of 555 px vertical, a major-to-minor ratio of about 3.1. Rectangularity $751878.4/(1726.7 \times 555.0) =$ **0.785**, which is exactly $\pi/4$: the ratio of an ellipse's area πab to its bounding-rectangle area $4ab$ is precisely $\pi/4 \approx 0.785$ — a number clean enough to read off the geometric identity. Circularity 0.321 is low because this is a 3:1 flat ellipse, far from a circle. Most notably, **the center and the gravity nearly coincide**: (1300.53, 950.50) versus (1300.53, 950.52), differing by less than 0.02 px — the standard part's outline is symmetric, so its centroid naturally lands at the geometric center; once a part carries an asymmetric defect, these two points separate, and **the gap between center and gravity is itself a probe**

of shape asymmetry (the registration section below will use this point).

These features relate to Blob features (Chapter 23) as a "boundary version" to a "region version": they ought to agree numerically, but the contour version is more accurate thanks to its subpixel point list and depends only on the boundary itself — when two targets touch into a single Blob on the region, their respective contours may still be cleanly separable. An honest note: the current SDK has **no curvature API**; when curvature is needed (e.g. to assess the corner quality of the ellipse ends), compute it yourself on the smoothed contour — the rate of change of the tangent angle of adjacent points divided by arc length (or a local circle fit on the point list) gives a discrete curvature estimate.

The definition of circularity varies among toolkits: some take $4\pi A/P^2$, others $A/(\pi r_{\max}^2)$ (with $r_{\max}$ the maximum distance from center to contour). The two definitions give different values for the same shape, so **comparing circularity across libraries is meaningless** — being self-consistent within one production line is enough.

24.3 Contour Operations

Before measurement and comparison, one often needs three kinds of preprocessing operations on the contour.

Smoothing: a sliding-window average of the point coordinates. With a window of 31, the master contour's perimeter shrinks from **3826.33 to 3820.25** (about 0.16%). The shrinkage is tiny, which precisely indicates that the backlit silhouette's boundary is clean enough — there is little high-frequency jitter to filter. But the direction of the shrinkage is fixed: noise makes the point list jitter back and forth around the true boundary, the jittered serrations are all high-frequency components, and the perimeter, which accumulates point distances, counts the length of every serration; smoothing is low-pass filtering (replaying on a 1D point list what Chapter 6 does on an image), removing high-frequency jitter so the perimeter can only decrease. Engineering corollary: **perimeter is systematically biased high by noise, so before measuring perimeter either smooth or fix the smoothing parameters to ensure comparability** — under a clean silhouette this bias is small, but in a noisy image a fixed smoothing policy is the prerequisite for comparability.

Resampling: reduce the 3627 points to a specified count; the experiment resamples to a target of 300 points, obtaining **300 points** approximately equidistant along the contour — to bound the cost of subsequent per-point computation, or to provide uniform samples for fitting. There is a parameter-semantics pitfall here, see Section 24.5.

Segmentation: cut the contour into line segments and circular arcs by geometric property. The master contour splits into **16 segments = 12 lines + 4 arcs** (Figure 24.3: line segments green, arc segments red, resample points as cyan ticks): the gentler arcs at the top and bottom of the ellipse are approximated by several short line segments (green), while the two high-curvature ends on the left and right are recognized as arcs (red) — this is exactly the sensible reading of "line-arc decomposition" for an ellipse: low-curvature regions as lines, high-curvature regions as arcs. The segment count and line/arc classification are sensitive to the smoothing amount and the two distance thresholds, so the parameters must be tuned to the feature size. The value of line-arc decomposition lies downstream: each line segment can be fitted for direction and position, each arc for center and radius, and the "contour" is upgraded into a dimensionable **geometric drawing** — paving the way for geometric measurement and robot path programming.

24.4 Contour Comparison: Defect Detection

The climax of the chapter: compare the test part's contour against the master's point by point, letting positions where the deviation exceeds the tolerance surface automatically. The principle goes in three steps — **registration**: transform the master contour into the test part's coordinate frame using a reference point and reference angle; **per-point nearest distance**: for each point of the test contour, find its nearest distance to the registered master contour; **threshold decision**: points whose deviation exceeds the tolerance `maxDis` are flagged as out of tolerance, and any out-of-tolerance point means NG. This is precisely the image-based inspection of the **profile tolerance** in GD&T: the actual contour is required to fall within a tolerance band of `maxDis` on each side of the theoretical contour.

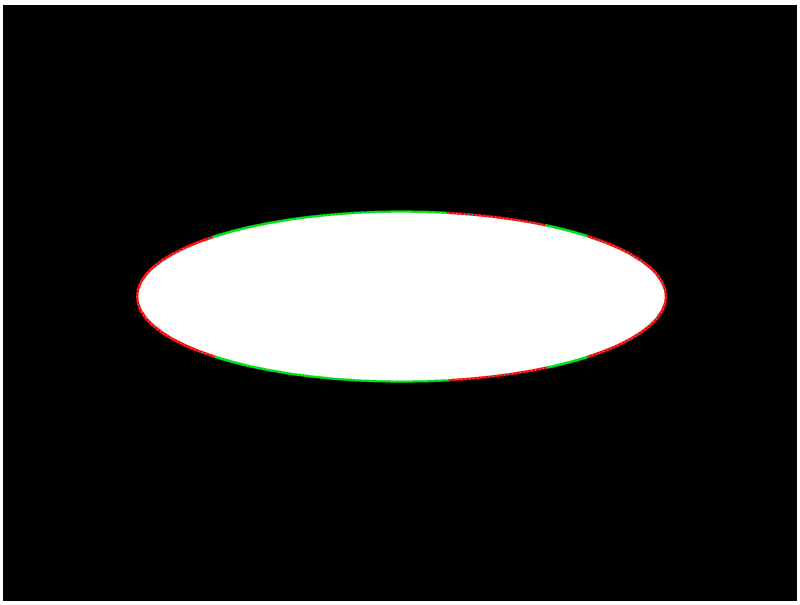


Figure 24.3: The result of operations on the master contour: after segmentation, line segments are green and arc segments are red (16 segments = 12 lines + 4 arcs, lines on the gentle regions and arcs on the high-curvature ends); the cyan ticks are the resampling to 300 approximately equidistant points.

The experiment uses `maxDis=5` px. The registration basis comes from **external localization**: integrate the zeroth- and second-order moments of each image's bright region to obtain its center and principal-axis angle (see below for why this is necessary). Comparing the top-defect part (`003`) against the master, the results:

- **Maximum deviation 18.70 px @ (1738.5, 695.4)** — the location is exactly the protrusion on the right of the top edge; the verdict is **NG**.
- The per-point mean deviation is **1.64 px** — this is the baseline fit of two independently imaged, independently registered subpixel contours over the full perimeter, about nine parts in ten thousand relative to the 1726.7 px major axis, the accuracy backbone of high-resolution silhouette extraction.
- Two defect segments are localized automatically: **segment #1 pts[418..514]** (peak 18.70 px, the right protrusion on the top edge), **segment #2 pts[3141..3232]** (peak 17.86 px, the left protrusion on the top edge). `GetOverThresholdContours` likewise outputs **2** out-of-tolerance contour segments.
- Switching to the bottom-bump part (`002`) against the master, the maximum deviation rises to **33.21 px @ (1554, 1250)**, mean 2.36 px, also NG — a larger bump yields a larger peak deviation.

Figure 24.4 rewards a color-by-color read: blue is the registered master contour — the "theoretical shape"; the test contour is colored by deviation, green 2 px, yellow 2–5 px, red >5 px; the red cross marks the maximum-deviation point. The two red clusters cover exactly the two protrusions on the top edge, and the rest of the contour is almost entirely green. Those two excess-material defects, which require a close look to discern by eye, have nowhere to hide on the deviation map — **contour comparison turns "finding the defect" into "reading a thermometer".**

Finally, the most important lesson of this section: **where does the registration reference point come from**. First consider a naive assumption that the real sample images puncture directly — "all three images are the same workpiece, so the

Per-point nearest distance is a one-directional measure: it only asks "how far is the test point from the master", not the reverse. If the test part is **missing** a stretch of contour (e.g. a chip-out), the test points all still lie close to the master — a strict shape distance (such as the Hausdorff distance) takes the two-directional maximum. In practice one often runs the comparison once in each direction to cover both kinds of defect.

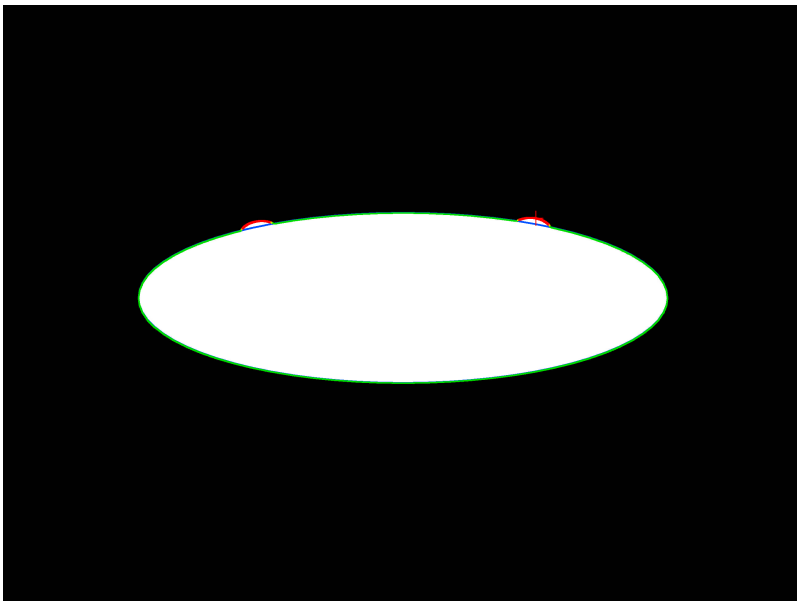


Figure 24.4: Contour comparison result. Blue: the registered master contour; the test contour is colored by deviation (green 2 px, yellow 2–5 px, red >5 px); the red cross is the maximum-deviation point at 18.70 px, located at the protrusion on the right of the top edge. The other protrusion on the left of the top edge is likewise captured by a red segment, two out-of-tolerance segments in all.

pose is of course the same, and an identity registration will do". The measured external localization vetoes it: the master's bright-region center is at (1300.00, 950.00), the top-defect part is at (1304.97, 954.02), off by **6.39 px**, and the bottom-defect part is at (1301.25, 951.38), off by **1.86 px**. Real capture has a positional drift each time the part is placed, so **registration is not an identity transform and must be performed**; forcing an identity registration would bake this 6.39 px global misalignment into every point and bury the defect signal entirely.

So what should the registration basis be? A seemingly handy approach is to use each contour's own **gravity** — free to compute, the defect part using the defect part's gravity. This chapter's probe gives an honest comparison: the top-defect part's contour gravity drifts 6.385 px relative to the master, almost exactly the 6.390 px measured by external localization — meaning the defect's **own contribution** to the gravity is only about 0.005 px, and the two bases are nearly equivalent here (maximum deviation 18.697 versus 18.698 px). The reason is simple: the defect is too small relative to the whole large ellipse (the excess material is only parts per million of the bright region), the gravity displacement is dominated by the part's real placement drift, and the defect cannot move it.

But **the discipline still holds, and is proportional to the defect size**: localizing the object by its own gravity is a systemic risk — the larger the defect, the more the excess (or missing) material drags the gravity off, the misalignment is read as a larger defect, and the measurement is contaminated by the measurand. Here the defect is small and the risk has not materialized; switch to a sample with a large burr and this path collapses. The robust approach is to take the reference point from **external localization**: a fixture, a locating pin, or moment integration / template matching on an **uninspected region** (the locating-datum problem discussed in Chapter 19 resurfaces here). This chapter uses moment integration over the entire bright region — which does two things at once: it corrects the 6.39 px real displacement, and, by covering a million pixels, it is naturally immune to local defects. The larger the defect, the more important this discipline.

24.5 SciVision Implementation

This chapter's pipeline spans four classes. Extraction uses `SciSvContourExtraction`:

```
SCIMV::SciSvContourExtraction ext;
SciContourArray mAll;
long rc = ext.ExtractContours(srcM, fullROI, /*precision subpixel*/0, /*method canny*/1,
    /*filterCoeff*/2.0, /*lowThreshold*/20, /*highThreshold*/40,
    /*minContourLength*/600, /*maxContourLength*/100000, &mAll);
```

`precision=0` outputs a subpixel contour; `method=1` is the Canny route, `filterCoeff` is its smoothing scale, and the two thresholds are the hysteresis thresholds; `minContourLength=600` is the filter threshold for stray contours. Note that `maxContourLength` has a valid range of **[0, 100000]** — passing a larger value (e.g. 1000000) raises a parameter error directly. Features use `SciSvContourFeature`, one call per feature:

```
SCIMV::SciSvContourFeature feat;
feat.GetContourArea(mC, &area);
feat.GetContourPerimeter(mC, &perim);
feat.GetContourCenter(mC, &center);
feat.GetGravityAndOrientation(mC, &gravity, &orient);
feat.GetContourCircularity(mC, &circ);
feat.GetContourRectangularity(mC, &rectg);
feat.GetSmallestRect(mC, &rcCenter, &rw, &rh, &rang);
```

Operations use `SciSvContourOperation`:

```
SCIMV::SciSvContourOperation op;
op.SmoothContours(mOne, /*window*/31, &smoothed);
op.SampleContour(mOne, /*method target point count*/1, /*sampleSize*/300.0, &sampled);
op.SegmentContours(mOne, /*smooth*/5, /*maxLineDistance1*/8.0f,
                   /*maxLineDistance2*/4.0f, &segs, &segType);
```

`SegmentContours`'s two distance thresholds control the allowed deviation of the line fit, and the output `segType`

labels each segment's type (0=line, 1=arc). Comparison uses `SciSvContourContrast`, with the reference point/angle coming from external localization (bright-region moments):

```
SCIMV::SciSvContourContrast cmp;
rc = cmp.ContrastContour(tOne, mOne, /*normRefPt*/refM, /*actRefPt*/refTest,
                         /*normAng*/angM, /*actAng*/angTest, /*maxDis*/5.0, /*outputMode*/0,
                         &maxPair, &maxOff, &results, &minPair, &minOff, &avgOff, &allOff);
SciContourArray overs;
cmp.GetOverThresholdContours(&overs);   // take the out-of-tolerance contour segments directly
```

There are two reference points/angles: `norm*` describes the master's pose, `act*` describes the test part's pose, and the registration transform is derived from their difference — here the two are measured independently from their own bright-region moments, thereby correcting the real placement drift. Pitfalls hit in practice (recorded in the engineering conventions):

- **Only `outputMode=0`'s scalar outputs are reliable.** At `outputMode=3`, `maxOff` is a meaningless value (measured −0.052) and `results` misreads OK; the per-point `allPointsOffset` is complete in all modes (here 3627 signed deviations).
- **`avgPointsOffset` is unreliable**: it returns 9.373 in measurement, whereas the true mean self-computed from `allPointsOffset` is 1.64 — compute the mean yourself.
- **`SampleContour`'s `method=0/3` is upsampling**, and only `method=1` downsamples by target point count — pass 300 intending to reduce points and you may receive more points instead.
- Some contour-related headers (`SciSvContourOperation`/`ContourContrast` etc.) **unconditionally `#define DLL_EXPORTS`** (an SDK slip); include them after the dllimport-style headers and they link normally.

The complete project that generates all the images and numbers in this chapter is in `code/contour_analysis/`, with the sample images in its `sample/` subdirectory.

Industry Case: Seal-Ring Burr Inspection

After a rubber seal ring is vulcanized and demolded, the burr remaining at the parting line is the main defect, and a typical inspection scheme is exactly contour comparison: take a silhouette under backlight, extract the ring's outer contour and compare it against a standard contour point by point, flagging out-of-tolerance as a burr — the same as the three ellipse samples of this chapter. One production line's first version took a shortcut and used the part's own gravity as the registration reference point — fine for small burrs (just as measured in this chapter, a small defect drags the gravity less than 0.01 px), but once a large burr appears, the excess material drags the gravity off, the registered standard contour shifts as a whole, and the **intact edge on the opposite side** is flagged out of tolerance over a large area, with "phantom defect" alarms firing frequently and re-inspection workload staying high. The revised scheme switched to the locating pin holes built into the mold as an external datum: locate the part's pose by two pin holes first, then run the contour comparison, and the false-alarm rate immediately dropped to negligible. The lesson matches this chapter's experiment exactly: the larger the defect, the less it can be allowed to take part in registration — the registration basis must be taken from external features unaffected by the defect.

24.6 Summary

- **A contour = an ordered string of subpixel boundary points**, the "boundary-version dual" of the Blob region representation: the region suits statistics and connectivity, the contour suits boundary deviation, piecewise geometry, and high-precision measurement. The extraction pipeline is "edge detection → linking → subpixel refinement", with the `minLen` length filter clearing stray contours; the backlit silhouette hands the crisp boundary to the subpixel extractor, and the three parts each yield a closed contour of 3627 points.
- **Contour features correspond one-to-one with Blob features but with higher accuracy**: area is integrated directly from the boundary by Green's theorem,

rectangularity 0.785 is exactly $\pi/4$ — self-proving this is a standard ellipse; the symmetric master's center and gravity nearly coincide (differing <0.02 px), and the gap between the two is a probe of shape asymmetry. Shape factors such as circularity are defined differently across libraries and cannot be compared across libraries.

- **Smoothing is low-pass**: a window of 31 takes the perimeter 3826.33→3820.25 — under a clean silhouette the shrinkage is only 0.16%, but its direction is fixed, so perimeter measurement must fix a smoothing policy; segmentation cuts the ellipse into 12 lines + 4 arcs (lines on the gentle regions, arcs on the ends), upgrading it into a dimensionable geometric description.
- **Contour comparison = registration + per-point nearest distance + tolerance decision**, the image realization of profile-tolerance inspection: with a baseline deviation of 1.64 px, this chapter's experiment cleanly captures the two protrusions on the top edge (peaks 18.70 and 17.86 px) and the bump on the bottom edge (peak 33.21 px), automatically localizing the defect segments and judging NG.
- **The registration basis is the make-or-break point**: the three real-sample workpieces are not co-located (external localization measures offsets of 6.39 and 1.86 px), so registration must be performed and the basis must be robust — this chapter uses moment integration over the entire bright region to correct the real displacement and stay immune to local defects at once; using the part's own gravity fails as the defect grows (here a small defect contributes only 0.005 px to the gravity, but a large burr would collapse it). Chapter 26 will generalize comparison-based defect detection to more general forms.

The classic algorithm for contour extraction (border following) is given by Suzuki and Abe (Suzuki and Abe 1985); the chain-code representation and processing of contours trace back to Freeman's survey of computer processing of line-drawing images (Freeman 1974); and the polyline-simplification idea behind this chapter's segmentation operation originates in the classic algorithm of Douglas and Peucker (Douglas and Peucker 1973). For a more systematic engineering treatment of subpixel

contour extraction, contour segmentation, and contour-based measurement, see the work of Steger et al. (Steger, Ulrich, and Wiedemann 2018).

25 Shape Features

After the past several chapters, we already hold quite a few tools that "turn images into numbers": blob analysis gives area and centroid (Chapter 23), contour analysis gives perimeter and circularity, edge detection gives position and direction. But the problems on a production line go far beyond these. For a meandering PCB trace, how should its "length" and "width" be defined? Two surfaces with exactly the same average brightness — one a uniform, acceptable coating, the other a reject covered in pits — what number tells them apart? When a board needs coarse localization, where do we find a few reference points that are "both sparse and stable"? This chapter gathers three tools — each independent, yet all answering "how to compress a shape into decidable numbers" — into one toolbox: the **skeleton** extracts the medial axis of elongated structures, **gray-level and texture features** extract a statistical fingerprint of surface appearance, and **corners** extract landmarks from geometric outlines. Figure 25.1 is the test scene for the skeleton experiments — a Y-shaped branching trace, an L-shaped trace, and a blob with a square hole, all typical samples that are 5 pixels wide or contain elongated/blocky structures.

25.1 Skeleton

Imagine placing the largest possible circle inside a binary shape and letting it roll around while always staying tangent to the boundary — the locus of all these **maximal inscribed circle centers** is the shape's **medial axis**; attach to each center the radius of its inscribed circle and you obtain the **medial axis transform**. Intuitively, the medial axis is the shape's "backbone": for a trace of constant width, the medial axis is exactly the centerline along the trace direction, and the inscribed-circle radius is exactly the half-width. This is the most natu-

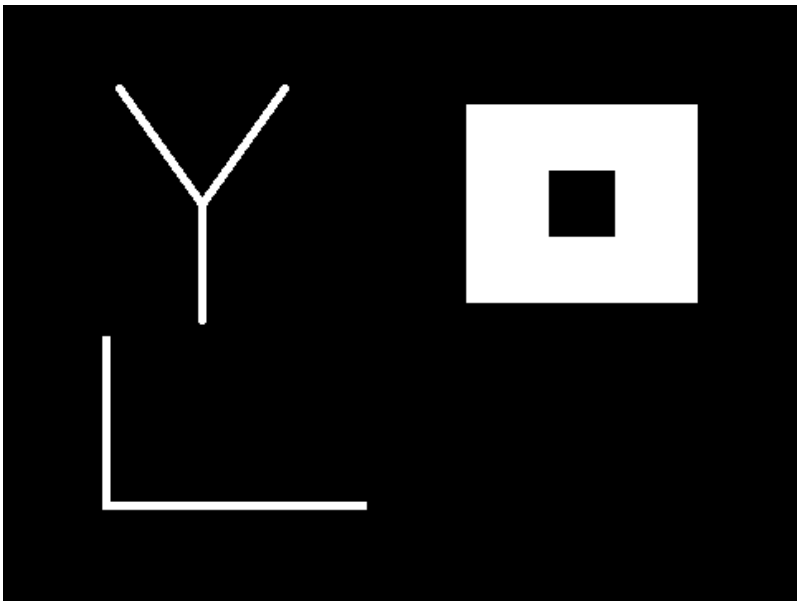


Figure 25.1: Binary test scene for the skeleton experiments: top-left, a Y-shaped branching trace (5 px wide); bottom-left, an L-shaped trace (5 px wide); top-right, a blob with a square hole.

ral description of an elongated structure — it reduces a two-dimensional "patch of pixels" back to a one-dimensional "line plus a width."

In practice the most common way to compute a skeleton is a **thinning** algorithm: iteratively peel pixels off the shape's boundary, while guaranteeing that no deletion ever changes connectivity — neither peeling one shape into two pieces nor peeling a hole out of existence — until only a single-pixel-wide centerline remains. It belongs to the family of morphological algorithms introduced in Chapter 8, and its output is a discrete approximation of the medial axis.

The medial axis is very sensitive to boundary noise: a small bump on the boundary makes the medial axis sprout an extra branch. Engineering implementations (including thinning algorithms) usually smooth the boundary first with the opening and closing operations of Chapter 8, or prune short branches (pruning).

Running SciVision's skeleton algorithm on Figure 25.1 gives the result in Figure 25.2. The three shapes produce 3 skeleton contours; the foreground of the entire scene contains 17729 pixels, while the skeleton retains only 738 — a **24.0-fold compression** — and yet none of the topology or course information of the elongated structures is lost. Note the skeleton of the holed blob at the top right: it forms a **closed loop** around the square hole, which is precisely the "topology-preserving" property of thinning at work — the shape has one hole, so the skeleton has one loop that can neither be broken nor shrunk away.

A skeleton is a structured curve, and the feature points on it carry engineering meaning of their own: an **endpoint** is where a trace starts or stops, and a **junction** is where it branches. Extracting feature points from the Y-shaped trace's skeleton yields 3 endpoints and 1 junction, in exact agreement with ground truth; the endpoints lie at (70, 51), (120, 190), (170, 51), and the junction lies at (120, 119), just 1 pixel from the geometric ground truth of (120, 120). For trace inspection these two kinds of points are the "circuit diagram": a wrong endpoint count means an open circuit, and a wrong junction count means a bridge.

The most important application pattern of the skeleton is a simple division:

$$\bar{w} = \frac{A}{L},$$

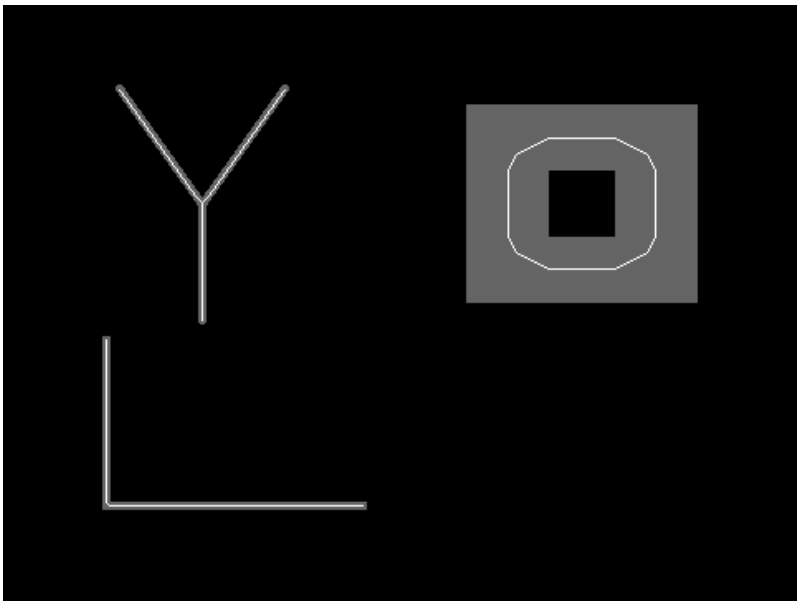


Figure 25.2: Skeleton result: the original shapes are shown darkened, with the single-pixel-wide skeleton overlaid in white. The Y-shaped trace's skeleton is three branches meeting at one fork; the holed blob's skeleton forms a closed loop around the hole.

where A is the foreground area of the trace, L is the skeleton length (pixel count), and $\bar{w}$ is the trace's **average width**. Measured on the L-shaped trace in the scene (designed width exactly 5.00 px): area 1313 px, skeleton length 257 px, average width $1313/257 = 5.11$ px. Where does the error come from? Skeleton length is counted as 4-connected pixels, and at the corner of the L the discrete count comes out slightly shorter than the geometric arc length (the corner gets "shortcut"); a smaller denominator naturally pushes the width slightly up. **A 2% systematic bias is entirely adequate for trend monitoring, but for absolute pass/fail width judgments you should calibrate with arc length (counting diagonal steps as $\sqrt{2}$) or with caliper measurement.** Honestly reporting this kind of discretization bias has more engineering value than pretending "measured = true."

The skeleton's typical uses can be listed in one inventory:

- **Line-width inspection**: area / skeleton length gives the average width; sampling the inscribed-circle radius point by point along the skeleton gives a width sequence;
- **Open-circuit and bridge detection**: count endpoints and junctions and compare against the design netlist;
- **Path planning and length measurement**: the "true length" of a meandering trace, a crack, or a fiber is its skeleton length.

25.2 Gray-Level and Texture Features

The question the second group of tools answers is: how does the **surface appearance** of a region become numbers? The first thing that comes to mind is first-order statistics — compute the gray-level histogram of all pixels in the region and take the mean, standard deviation, and entropy. They are cheap and intuitive, but they have one fundamental blind spot: **first-order statistics see only "which gray values are present," and are completely blind to "how those gray values are arranged."**

The four 100×100 texture regions in Figure 25.3 were designed precisely to expose this blind spot: uniform gray 140, a high-

contrast checkerboard (alternating 40/240), a horizontal gradient (40→240), and Gaussian noise ($\mu = 140$, $\sigma = 20$) — the **means of all four regions are approximately 140** (measured 140.00 / 139.84 / 139.51 / 139.76). By the mean, the four regions are one and the same. The standard deviation opens up some gaps (0 / 100.00 / 58.18 / 20.04), but look at the entropy: 6.64 for the gradient region, 6.36 for the noise region — nearly identical. In the eyes of first-order statistics, a "smoothly transitioning gradient" and "scrambled noise" are hard to tell apart, because both spread their histograms wide.

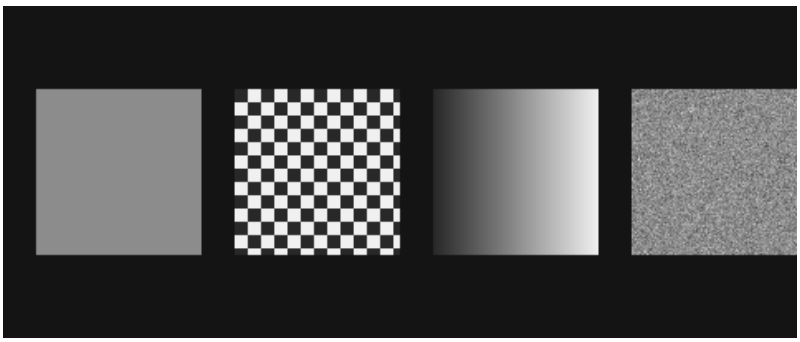


Figure 25.3: Four texture regions with mean approximately 140 (left to right): uniform 140, high-contrast checkerboard, horizontal gradient, Gaussian noise $\sigma = 20$. First-order statistics cannot fully distinguish them; the distinction rests entirely on second-order statistics.

To see "arrangement," look at **pixel pairs**. The **gray-level co-occurrence matrix (GLCM)** counts, for a given direction and distance (most commonly: horizontally adjacent), the frequency $P(i, j)$ with which a pixel of gray level i has a neighbor of gray level j — it is the **joint distribution** of the gray levels of adjacent pixel pairs. After quantizing the gray values into a small number of levels (16 in this experiment), the matrix is not large, yet the texture's "arrangement law" is written entirely inside it: a uniform texture concentrates its energy in a single cell on the diagonal, a gradient texture concentrates in a narrow band near the diagonal, and a noise texture spreads across the whole matrix. From $P(i, j)$ four common scalars are distilled:

- **Energy** $\sum_{i,j} P(i,j)^2$: the more concentrated the distribution (the more uniform and regular the texture), the larger the value; 1 for a uniform region;
- **Contrast** $\sum_{i,j}(i-j)^2 P(i,j)$: the larger the gray difference between adjacent pixels, the larger the value — a measure of "local contrast";
- **Correlation**: the statistical correlation coefficient between i and j — the degree to which adjacent pixels "know each other";
- **Homogeneity** $\sum_{i,j} \frac{P(i,j)}{1+|i-j|}$: the closer the mass lies to the diagonal, the larger the value — a measure of "local smoothness."

The full measured table for the four regions is below (mean/standard deviation computed ourselves, for the reason given in Section 25.4; GLCM taken in the horizontal direction with 16 gray levels):

Region	Mean	Std. dev.	Entropy (1st-order)	GLCM energy	Contrast	Correlation	Homogeneity
Uniform 140	140.00	0.00	0.00	1.000	0.000	0.000	1.000
Checkerboard	139.84	100.00	1.00	0.393	20.485	0.758	0.879
Gradient	139.51	58.18	6.64	0.061	0.131	0.996	0.934
Noise $\sigma = 20$	139.76	20.04	6.36	0.048	3.350	−0.015	0.467

The GLCM has a **direction**: this experiment counts horizontally adjacent pairs. For isotropic textures (noise, uniform), the direction does not matter; for anisotropic textures (scratches, brushed finishes, the horizontal gradient in this example), switching direction changes the numbers completely — counting the gradient region in the vertical direction would yield a contrast of 0. In practice, either choose the direction from the known orientation of the defect, or average over the four directions.

Read the table row by row and ask "why this signature." The **uniform region** is the degenerate case: only one gray level, so energy 1, homogeneity 1, contrast 0, and correlation undefined because the variance is zero (the SDK returns 0). The **checkerboard region**'s signature is high contrast (20.48, two orders of magnitude above the lowest non-trivial entry in the table): with 8 px squares, about 1/8 of the horizontally adjacent pairs straddle a black–white boundary, and every crossing contributes an enormous $(i-j)^2$; the remaining 7/8 of the pairs share a square and a gray level, which is why the correlation still reaches 0.758

— "violent contrast, but arranged with regularity." The **gradient region**'s signature is high correlation (0.996) paired with low contrast (0.131): horizontally adjacent pixels differ by only about 2 gray levels and are almost perfectly predictable from one another — "large change, but change with order." The **noise region**'s signature is low homogeneity (0.467, the lowest in the table) plus zero correlation (-0.015): adjacent pixels are mutually independent — neither can predict the other. Four regions with identical means **each occupy their own corner** in the four GLCM quantities, cleanly separable — and that is the value of second-order statistics: they turn "arrangement" into numbers.

25.3 Corners

The third group of tools hunts for geometric **landmarks**. What makes a corner special can be understood through the "self-similarity" of a small window: shift the window a small step in any direction and see how much its content changes. In a flat region, no shift in any direction changes anything; on an edge, shifting along the edge changes nothing while shifting perpendicular to it changes a lot — only one direction is "sensitive"; but at a corner, **a shift in any direction changes the content drastically** — there are strong gradients along two independent directions. The Harris corner detector (Harris and Stephens 1988) turns this intuition into an eigenvalue criterion on the gradient structure matrix within the window: only when both eigenvalues are large is it a corner.

The experimental scene is an L-shaped hexagonal plate (6 right angles, 1 of them concave) plus a distractor — a rounded rectangle with corner radius 25 px, which has **no sharp corners at all**. Extracting corners with the Harris method at a filter ratio of 0.30 gives the result in Figure 25.4a: exactly 6 corners detected, in one-to-one correspondence with ground truth, each corner sitting 1.41 px from its geometric truth — a consistent $(\pm 1, \pm 1)$ bias. That the bias is so tidy is no coincidence: the Harris response peaks where the gradient structure is richest, and for a right angle evaluated over a 5×5 neighborhood (blockSize=5), that peak lands stably one diagonal pixel from

the corner. **A systematic bias can be calibrated and compensated; random jitter is the true killer of accuracy** — and this 1.41 px belongs to the former.

(a) filterRatio = 0.30: exactly 6 corners, the crosses landing precisely on the 6 right angles of the L-shaped plate (including the concave one), with no detections on the rounded rectangle

(b) filterRatio = 0.02: 14 "corners," 8 of which land on the four rounded corners of the rounded rectangle

Figure 25.4: Threshold sensitivity of Harris corner extraction. Strict filtering keeps only true sharp corners; loose filtering lets gentle rounded corners produce enough Harris response to be detected.

Lowering the filter ratio from 0.30 to 0.02 makes the detection count explode from 6 to 14 (Figure 25.4b), 8 of which land on the four rounded corners of the rounded rectangle — each rounded corner detected as 2 "corners." This reveals a fact that is often overlooked: **corner "strength" is a continuous quantity; "whether it is a corner" is an engineering definition.** At a rounded corner the curvature is nonzero, there really are gradients in both directions, and the Harris response really is greater than zero — just weaker than at a right angle. Wherever you set the threshold, that is where the boundary of "corner" lies. This is the same philosophy as the threshold tuning of Chapter 13: the algorithm outputs a response map, and the "decision" is always made by a human on the algorithm's behalf. The tuning method is the same too — scan the threshold on samples containing known distractors and find the safe interval of "all true corners detected, zero rounded corners detected."

What do you do once the corners are found? Three typical uses: use 2–3 stable corners for **coarse localization** of a workpiece (much faster than template matching); the grid intersections of a calibration plate are corners in essence, the input to **camera calibration**; and as interest points for feature matching — the first internal step of the feature matching in Chapter 18 is exactly the detection of these "changes no matter which way you shift" points.

25.4 SciVision Implementation

The skeleton is provided by `SCIMV::SciSvSkeleton`, with two interfaces used together:

```
SCIMV::SciSvSkeleton sk;
SciContourArray arr;
sk.Skeleton(src, roi, &arr);          // thinning -> array of skeleton contours
SciPointArray ends, juncs;
sk.ExtractJunctions(arr[i], &ends, &juncs);  // endpoints/junctions of a single contour
```

`Skeleton` thins every connected shape inside the ROI into a skeleton and outputs a contour array — note that it **does not split at junctions**: the entire skeleton of the Y-shaped trace is one contour, not three branches. This is deliberate engineering design: whether to split, and where, is decided by `ExtractJunctions` — it takes a single skeleton contour and outputs an endpoint array and a junction array, and the caller then splits at the junctions as needed. The two-stage "thinning is thinning, parsing is parsing" arrangement lets one API serve both the user who only wants a length and the user who wants the full topology.

Gray-level features are provided by `SCIMV::SciSvGreyFeature`:

```
SCIMV::SciSvGreyFeature gf;
double ent, aniso;
gf.GetEntropyGreyFeature(src, roi, &ent, &aniso);   // first-order entropy + anisotropy
double ge, en, corr, homog, contr;
gf.GetGLCMFeature(src, roi,
```

```
    SCI_SV_GREYFEATURE_GLCM_DIRECTION_HORIZONTAL,    // direction: horizontally adjacent pairs
    16,                                              // number of gray quantization levels
    &ge, &en, &corr, &homog, &contr);                // GLCM entropy/energy/correlation/homoge
```

`GetGLCMFeature`'s `greyClass=16` is the quantization level count: more levels mean a larger matrix and more sensitivity to noise; 16 is the common compromise. For the meaning of the direction parameter, see the margin note in Section 25.2. One reminder about an old pitfall: the `stdValue` of `SciSvHistogram::CaculateHist` is the standard deviation of the histogram bin counts, **not** the gray-level standard deviation, so the means and standard deviations in this chapter's table were computed from the definition ourselves.

Corners are provided by `SCIMV::SciSvCornerExtraction`:

```
SCIMV::SciSvCornerExtraction ce;
SciPointArray pts;
ce.ExtractCorners(src, roi,
    0,        // method: 0 = Harris
    5,        // blockSize: evaluation neighborhood of the gradient structure matrix
    50,       // maxnumCorners: upper bound on count
    0.30,     // filterRatio: response filter ratio (the protagonist of this chapter's sensiti
    10,       // minDistance: minimum distance between corners, suppresses duplicate detection
    &pts);
```

`minDistance=10` explains why each rounded corner yields "only" 2 detections under the weak threshold — without it, detections would pop up along the entire arc. The complete project that generates all of this chapter's images and data is located at `code/shape_features/`.

Industry Case: Open-Circuit Detection on Flexible PCBs

An FPC (flexible printed circuit) production line inspects trace health after etching. The initial scheme used blob connectivity analysis: a trace broken into two blobs was reported as an open circuit. It could only catch "fully broken," while the **necking** caused by over-etching (a trace locally thinned but not yet severed) was missed entirely — and the necked parts failed in batches during bend testing. The improved scheme

introduced the skeleton: extract the skeleton of each trace and compute, in a sliding window along the skeleton, "local area / local skeleton length," yielding a **width sequence along the trace**. The width sequence of a healthy trace is flat; a neck appears as a clear valley in the sequence — set a lower width limit, and a half-broken trace is intercepted before it breaks. The lesson: the skeleton reduces "two-dimensional shape health" to a "one-dimensional signal," and on a one-dimensional signal the classic threshold, trend, and SPC tools all become immediately applicable — **dimensionality reduction is not losing information; it is placing the information within reach of the old tools.**

25.5 Summary

- **The skeleton is the medial axis of elongated structures**: thinning iteratively peels the boundary while preserving topology (holes become closed loops), compressing a 17729 px foreground into a 738 px centerline (24-fold) without losing course or topology; endpoints/junctions map directly to open-circuit/bridge criteria.
- **Width = area / skeleton length** is the skeleton's most practical pattern (measured 5.11 vs ground truth 5.00); pixel counting underestimates arc length at corners, introducing about a 2% systematic bias — negligible for trend monitoring, but absolute judgments require calibration.
- **First-order statistics cannot see "arrangement"**: four texture regions all with mean 140 cannot be fully distinguished by mean and entropy; the GLCM turns the joint distribution of adjacent pixel pairs into the four numbers energy/contrast/correlation/homogeneity, giving each region a unique signature (checkerboard = high contrast, gradient = high correlation with low contrast, noise = low homogeneity with zero correlation).
- **Corner strength is a continuous quantity; "whether it is a corner" is an engineering definition**: filterRatio=0.30 detects exactly 6 corners with a consistent 1.41 px bias (calibratable); loosening to

0.02 lets the rounded-corner distractor contribute 8 false detections — threshold tuning must scan samples that contain distractors.

- What the three tools share is **compression**: squeezing a patch of pixels into a line, a distribution, or a few points is what gives downstream decision logic something to hold on to.

The original definition of the medial axis comes from Blum's shape-description transform (Blum 1967), the theoretical parent of this chapter's skeleton; the canonical source of GLCM texture features is Haralick et al.'s foundational paper on texture-based image classification (Haralick, Shanmugam, and Dinstein 1973); the Shi–Tomasi corner criterion (Shi and Tomasi 1994), a sibling of the Harris corner (Harris and Stephens 1988) and widely used for coarse workpiece localization, is given there; and for pose-invariant shape descriptors, Hu's moment invariants (Hu 1962) remain an enduring classic. For the rigorous definitions of the skeleton and the medial axis transform, and a systematic engineering discussion of GLCM texture features, see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

26 Defect Detection

The previous three chapters of the inspection part handed you three tools: blob analysis (Chapter 23) segments and measures regions from a binary image, contour analysis (Chapter 24) reconciles an actual contour against a theoretical one point by point, and shape features compress "what it looks like" into comparable numbers. But all of them answer "what is this thing, how big is it, how far is it off" — while the most common question on a production line is a different one: "does this product **have anything it shouldn't have**?" Contamination, foreign objects, scratches, edge chips, burrs… defect morphologies can never be enumerated exhaustively, and you cannot write a detector for each one. This chapter presents the **paradigm** that closes out the entire inspection part: **a defect is a deviation from "normal."** The problem is thereby inverted — rather than defining the endlessly varied defects, first define the one and only "normal." Following the thread of "where does normal come from," industrial defect detection splits into three routes: **reference comparison** — build a reference from good samples and compare pixel by pixel (the variation model, this chapter's protagonist); **rule-based judgment** — describe the acceptance condition with geometric rules, such as "edge straightness must not exceed 2 px" (edge-defect and contour-defect detection); **statistical outlier detection** — rely on no external reference at all and find statistically anomalous regions within the image itself. The chapter runs all experiments on a set of **real industrial sample images**: 5 frames of 2448×2048 8-bit grayscale from Smart3's "scratch-inspection example recipe" (Figure 26.1), each a fibrous filter disk (disk surface gray level about 150, dark background about 80), with real scratches scattered across the surface and each carrying one class of real defect — a large dark contamination, a bright foreign object, an edge chip. It is precisely this set of "no clean master, only real parts" samples that forces out several lessons

the variation model never gets to learn in an idealized synthetic scene.

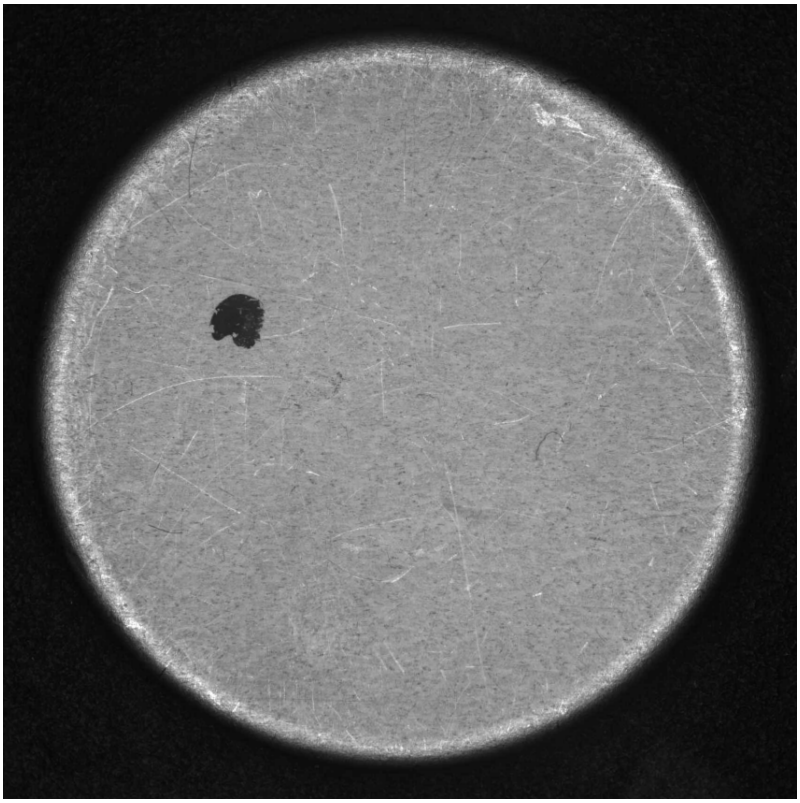


Figure 26.1: One real defective filter disk (sample 002) from the "scratch-inspection example recipe": the fibrous surface is covered with real scratches, and upper-left of center there is a large dark contamination at near-background gray level (an embedded foreign object / missing material). The disk is bright at the center with a bright rim halo; the surface is intensity-normalized to a mean of 150.

26.1 The Variation Model: Let the Samples Define Normal

The naive approach to filter inspection is to subtract a golden-sample image from the image under test and threshold the difference. It fails immediately: the disk surface is a **random fiber texture**, and the fiber orientation differs on every disk; even between two good parts, subtracting pixel by pixel sprouts "defects" at every fiber and every scratch. The insight of the **variation model** is this: normal is not a single image but a **band**; and the width of the band should not be dictated by an engineer's gut feeling — it should be **set by the variation of the training samples themselves**.

For n good training images, compute the per-pixel gray-level mean $\mu(x, y)$ and standard deviation $\sigma(x, y)$; the acceptance interval for that pixel — the **tolerance band** — is then

$$\big[\,\mu - \max(a,\ k\sigma),\quad \mu + \max(a,\ k\sigma)\,\big],$$

where a is the absolute threshold and k is the variation threshold coefficient. The $k\sigma$ term lets the tolerance adapt to each pixel's true variation; a is the floor — in regions where σ is close to zero, a σ estimated from only a few samples is itself unreliable, and the absolute lower bound keeps the tolerance band from collapsing to zero width and flagging pure noise as defects. At inspection time, any test pixel that falls outside the band is marked as a defect pixel. The model itself is just two images: an upper-bound image and a lower-bound image.

But **this set of real samples drives the variation model's prerequisites straight into a wall**. The textbook variation model carries a hidden assumption: the n good images must be **pixel-wise repeatable** — the same coordinate sees the same structure on every good part, and the only difference comes from process variation (registration jitter, ink quantity, illumination). A printed label satisfies this; a loose fibrous filter disk **does not**: register the 5 disks by centroid (translate to center, normalize intensity) and train the SDK variation model, and the texture simply does not line up at the pixel level — the per-pixel σ is blown up by texture misregistration, and the result

reports nearly a thousand texture false-alarm regions on **every** disk, including the cleanest golden sample (Section 26.2 gives the measured 692–1459). This is the first lesson the real samples teach: **the registered golden-sample variation model needs a "repeatable appearance," and it cannot handle non-repeatable textures such as fibers, fabric, or matte surfaces**.

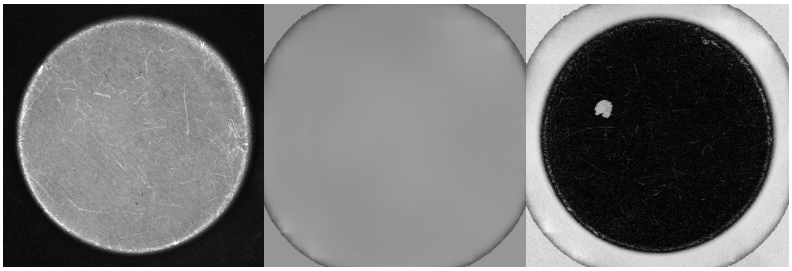
The way out is to change where "normal" comes from: not across samples, but **within the same image**. For each disk, its own surface is slowly varying at the large scale (bright center, rim halo, overall flat field), and **local anomalies** are the defects. So the "normal model" μ is taken as that disk's own **large-window smooth background** (a disk-masked box mean of radius 150 px), which erases scratches and contamination and keeps only the slow illumination; the σ in the band width is taken as the **robust scale of the golden sample's residual**, $1.4826 \cdot \mathrm{MAD} = 8.69$ (the normal noise level of the disk's fiber texture), with a floor $a = 10$. Detection then lands on the residual $R = I - \mu$: $|R| > \max(a, k\sigma)$ is a defect. This is still the same variation-model band mathematics, only with μ from spatial smoothing and σ from the golden texture — it sits at the intersection of the "reference comparison" and "statistical outlier" routes.

"Repeatable appearance" is the invisible threshold of the reference-comparison route. Print, labeling, screen printing, and molded-part surfaces are pixel-wise repeatable after registration — the variation model's home turf — while fiber paper, nonwovens, brushed metal, and frosted surfaces are pixel-wise random, and cross-sample registration only bakes texture misregistration into the tolerance band. To judge whether a surface can use a registered variation model, ask one question first: register two good parts and subtract — is the difference image flat, or covered in texture?

One engineering trick is worth pointing out: disk segmentation uses Otsu, and **the core of the dark contamination is darker than the threshold, so it is assigned to the background, leaving a hole in the disk mask**. The smooth background therefore takes the surrounding bright fiber values at the hole (in the mean image of Figure 26.2b the contamination has been "filled bright" into a clean surface), so the residual at the hole is strongly negative — the contamination hole is cleanly exposed by this very mechanism as a large negative-residual defect (the variation image of Figure 26.2c).

26.2 Detection Experiments

First put the measured **failure of the registered multi-sample SDK variation model** on the table. Take the three



(a) Golden disk (005, cleanest)

(b) Self-reference "normal" background μ

(c) Variation / residual image $|R|$

Figure 26.2: The self-reference variation model. (a) Golden disk 005, with only fine scratches left on the surface, used to calibrate the normal texture noise scale $\sigma = 8.69$; (b) the large-window smooth background μ of defective disk 002 — scratches and contamination are flattened away, and the contamination hole is "filled bright" by the surrounding fibers into a clean surface, exactly what "normal" should look like; (c) the residual image $|R|$: the surface is nearly all black (normal), while only the dark contamination upper-left lights up as a whole blob and fine scratches surface as faint bright lines; the bright rim halo also lights up — the latter is excluded by the `interiorR` circle.

disks with clean interiors — 003, 004, 005 — as good references (their real defects are at the edge and are excluded by an interior circular mask), register by centroid and normalize intensity, then train the SDK variation model ($a = 12$, $k = 3$) and compare disk by disk:

Table 26.1: Output of the registered multi-sample SDK variation model on the fibrous surface: every disk — including the golden sample 005 used for training itself — is flooded with nearly a thousand false-alarm regions by texture misregistration, with the real defects drowned among them; unusable.

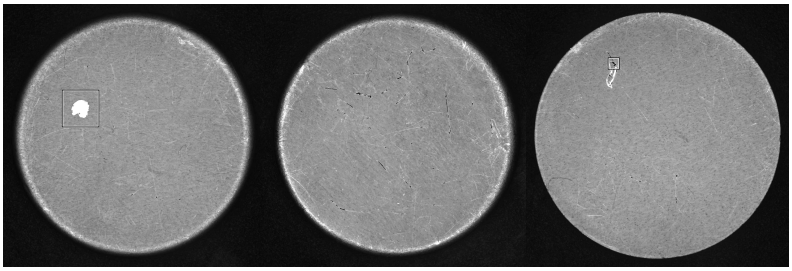
Test disk	001	002	003	004	005 (golden)
Defect regions	1017	1298	692	1459	1094

Golden sample 005 is one of the model's own training images, yet it is still reported with 1094 "defects," which nails the problem down: it is not a poorly tuned threshold but the fact that **the prerequisite of pixel-wise repeatability does not hold**. Switching to the self-reference band of Section 26.1, detection immediately becomes readable. At the normal setting ($k = 5$, band width $\max(10, 5 \times 8.69) \approx 43$, minimum connected component 30), detecting disk by disk and counting separately by polarity (dark defect / bright scratch):

- **002 dark contamination**: detected dark-defect area **9811** pixels, largest connected component **9738** pixels, mean contrast **−101** gray levels — a contamination blob nearly 10,000 pixels in size and 100 levels darker than the surface, at position (504, 676) (Figure 26.3a). It is the strongest, most unambiguous real defect of the whole set.
- **001 foreign object**: a curled bright fiber object embedded on the surface — the bright part detected (largest connected component contrast **+93** gray levels) and its dark shadow detected at the same time (dark-defect area **2960**, largest 2771, contrast −82); both polarities cross the band (Figure 26.3c).
- **005 golden sample**: dark-defect area only **343** pixels (no contamination), but the real scratches on the sur-

face are detected as **2934** bright pixels (Figure 26.3b) — "clean" for a golden sample is only relative; it too is covered with scratches.

- **003 / 004**: interior dark-defect areas 480 / 440 (minor); their real defect is an **edge chip**, which falls outside the interior detection circle and is handled by the rule-based route (Section 26.4).



(a) 002 dark contamination (b) 005 surface scratches (c) 001 bright foreign object

Figure 26.3: Detection results at the normal setting ($k = 5$), with dark defects overlaid in inverted white and bright defects in inverted black, and the strongest region boxed. (a) The dark contamination blob of 002 (9738 pixels, contrast -101) is cleanly enclosed; (b) the real scratches on the surface of 005 are detected as strings of bright lines — a golden sample has scratches too; (c) the curled bright foreign object of 001 and its dark shadow are detected with both polarities.

Case 002 confirms the robustness of the self-reference route: the contamination blob is a hundred gray levels darker than the surface, far from the noise tail, and crosses the band solidly regardless of how tight or loose the threshold is. The scratches of 005, on the other hand, demonstrate the other extreme — they are only ten to a few tens of gray levels brighter than the surface, hugging the noise tail, the real battleground of threshold tuning.

26.3 The Threshold Dilemma

The normal setting of $k = 5$ did not fall from the sky. Sweeping the same self-reference model and the five disks across tight ($k = 3$, band 26), normal ($k = 5$, 43), and loose ($k = 8$, 70) settings (floor $a = 10$, minimum connected component 30), the detected areas split by polarity are as follows (hand-written deterministic band, cross-validated against the SDK output in Section 26.5):

Table 26.2: Detected areas under three threshold settings (pixels, minimum connected component 30). Dark defects (002 contamination, 001 foreign object) pierce through all thresholds; bright scratches and grains fade rapidly with the threshold — the bright detections of golden sample 005 fall from 13384 at the tight setting all the way to 236 at the loose setting.

Disk (defect)	Dark tight	Normal	Loose	Bright tight	Normal	Loose
001 bright foreign object	5939	2960	**1935**	4984	864	179
002 dark contamination	12065	9811	**9133**	4592	241	0
003 edge chip	5228	480	0	8729	1552	74
004 edge chip	4168	440	109	13345	1520	163

Disk (defect)	Dark tight	Normal	Loose	Bright tight	Normal	Loose
005 golden sample	2204	343	35	**13384**	2934	**236**

The differing fates of the two defect classes in the table are precisely the fundamental dilemma of defect detection. **Dark contamination and foreign objects pierce through all thresholds**: the dark defect of 002 barely changes from 12065 at the tight setting to 9133 at the loose, and the 001 foreign object holds from 5939 to 1935 — their contrast is strong enough that they are the "must-catch" real defects. **Scratches and grains, by contrast, are extremely threshold-sensitive**: the bright detection of golden sample 005 collapses from 13384 pixels at the tight setting to 236 pixels at the loose, and the bright scratches of 003 and 004 also decay by orders of magnitude. Figure 26.4 puts the two extremes of 005 side by side: in the left image (tight) the surface is densely covered with detected fine scratches and fiber grains — if scratches are an acceptable cosmetic blemish, these are **false alarms**; in the right image (loose) the frame is nearly clean, leaving only the deepest few scratches.

This is exactly where real samples are more honest than a synthetic scene: **there is no "clean good part" here** — every disk has scratches, and the boundary between "defect" and "blemish" is decided by the acceptance criterion, not by physics. The engineering procedure for tuning is a **squeeze from both ends**: first, on a set of disks judged "acceptable," tighten the threshold until acceptable blemishes (shallow scratches) just stop being reported and false alarms are pressed to the permitted PPM level — this gives the lower bound; then verify detection on the **known must-catch real defects** (contamination, foreign object, chip) — this gives the upper bound. Between the two bounds lies the working window — and if the window does not exist (the shallowest must-catch defect and the acceptable blemish are inseparable in gray level), what needs

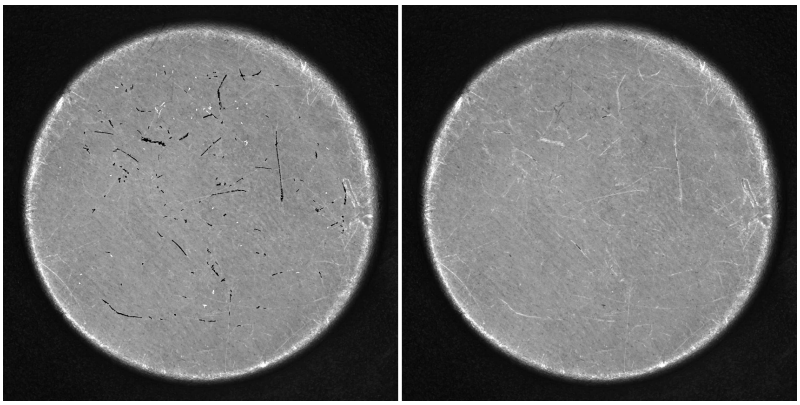


Figure 26.4: The threshold dilemma (both test images are golden disk 005). Left: the tight setting ($k = 3$) detects 13384 pixels, the surface densely covered with fine scratches and fiber grains; right: the loose setting ($k = 8$) leaves only 236 pixels of the deepest scratches. Whether a scratch is a "blemish" or a "defect" is decided by the acceptance criterion — and the threshold is exactly where that line sits in gray level.

changing is not the threshold but the imaging setup: lighting angle, spectral band, or resolution (the dark-field/bright-field choice of Chapter 4 is often made precisely to pry this window open).

26.4 The Rule-Based Route: Edge and Contour Defects

This is isomorphic to the lesson of Chapter 7: a threshold's "automatic" merely automates the act of picking a threshold and does not guarantee a fit to your scene. The variation model's "automatic tolerance" is the same — $k\sigma$ automatically adapts to the texture noise **present in the golden sample**, but "which deviation counts as a defect" is still an artificial line that must be drawn by the acceptance criterion, and "automatic" is no exemption from inspection.

The self-reference variation model excels at contamination, foreign objects, and scratches **inside the disk surface**, but it cannot reach the **edge chips** of 003 and 004 — they fall outside the interior detection circle and are essentially the disk's **contour** deviating from the circle it should be. When the acceptance condition can be **written down as geometric rules** (for instance, "the edge should be a circle of radius 760"), SciVision provides two modules on the rule-based route, introduced here as reconnaissance (not demonstrated in this chapter).

Single-edge defect detection (`SingleEdgeDefectDetection`, manual 8.11) targets requirements of the form "an edge should be straight/smooth": along a set of search lines it extracts point pairs from the standard edge and the actual edge, computes the distance and angle deviation for each pair, and judges NG when a threshold is exceeded. It is essentially caliper-style (Chapter 20) line-by-line measurement plus limit checking, suited to **local single-edge anomalies** such as burrs on stamped parts, chipped blade edges, and nicks in sealing strips — exactly matching the edge chips of disks 003 and 004 in this set: take the disk's theoretical circle as the standard edge, and where the actual edge dents inward beyond tolerance is a chip, no golden sample needed.

Contour defect detection (`ContourDefectDetection`, manual 8.14) compares the actual contour against a template contour as a whole and, beyond out-of-tolerance flagging, provides **defect classification**: breaks, dents, bulges, and fine burrs. It also supports width-defect detection between contour pairs and can generate a dispensing trajectory from the template — a set of capabilities plainly aimed at **glue dispensing/coating**

inspection: whether the glue path has gaps (breaks), glue pile-up (bulges), insufficient glue (dents), and whether the glue width is in spec (contour-pair width). It belongs to the same family as the contour comparison (`ContrastContour`) of Chapter 24: the latter outputs a continuous deviation curve, the former directly outputs a classified defect list.

The division of labor between the two routes fits in one sentence: **with a geometric prior and writable rules, take the rule-based route; with repeatable samples but no writable rules, take the variation model**. The "normal" of the disk's fiber texture cannot be described by a few geometric constraints, nor reproduced by cross-sample registration, and can only rely on single-image self-reference statistics; conversely, the disk's edge "should be a circle of radius 760" is one clean rule, and there is no reason to train a model for it.

26.5 SciVision Implementation

The variation model is provided by `SCIMV::SciSvVariationModel`, with one interface each for training and comparison:

```
SCIMV::SciSvVariationModel vm;
SciImageArray trainImgs;          // good training images (must be pixel-wise repeatable)
SciROIArray   maskArr;            // per-image mask regions (GenCircle for the disk interior)
SciROI        trainRoi;           // training region
SciImageArray model;              // output: [0] upper bound 8U [1] lower bound 8U [2] reserve
long rc = vm.TrainModel(trainImgs, maskArr, trainRoi,
                        /*absThreshold*/ 12.0f, /*varThreshold*/ 3.0f, &model);

SciImage result;  SciContourArray contours;
SciVarArray lengths;  SciPointArray centers;
SciROI roi;                       // UNDEF = compare the whole image (circular ROI rejected, s
rc = vm.CompareModel(testImg, model, roi, /*minLength*/ 30, /*maxLength*/ 100000,
                     &result, &contours, &lengths, &centers);
```

`absThreshold` and `varThreshold` are the a and k of the formula in Section 26.1. The key finding of this chapter on the real samples is that **registered multi-sample training is unusable on the fibrous surface** (Table 26.1; the golden sample

itself reports 1094 false-alarm regions). The way out is to treat `CompareModel` as a generic **"out-of-band pixel → contour extraction → length filtering" engine**, feeding it a **per-image self-synthesized self-reference model** — `model[3]` holds that disk's smooth background μ, `model[0]/[1]` hold $\mu \pm \max(a, k\sigma)$, and `model[2]` is zeroed. This both sidesteps the registration prerequisite and reuses the SDK's contour and length filtering. Measured cross-validation (band width 43, $k = 5$):

Table 26.3: Per-disk cross-validation of the SDK `CompareModel` engine (self-reference synthesized model) against the hand-written band: the total defect-pixel counts agree (002: 9926 vs 10052), differing only in region granularity due to different connectivity rules.

Disk	SDK comparison output (interior nonzero px / regions)	Hand-written band (area / regions)
002 contamination	9926 / 100	10052 / 9
001 foreign object	3666 / 46	3824 / 18
005 golden sample	2768 / 184	3277 / 54

This family of interfaces exposed several pitfalls in testing that must be recorded faithfully:

- **`minLength` must be 1**: passing 0 returns error 120001014, even though the documentation states a valid range of [0, 100000].
- **`maxLength` must be 100000**: passing 1000000 returns error 120001015 (counter to the intuition of an "upper bound on length filtering," and undocumented) — a new pitfall measured in this chapter.
- **All four output parameters must be given real objects**: passing `NULL` for any one of them crashes outright (0xC0000005), even if you do not care about the contours or centers.

- **`CompareModel` does not accept a circular ROI**: passing an ROI generated by `GenCircle` returns 120001015 (whereas `TrainModel`'s mask accepts circular ROIs); the comparison can only use a UNDEF/rectangular ROI, and restricting to the disk interior must be done by post-processing with a circular mask.
- **Registered multi-sample training fails on non-repeatable textures**: see Table 26.1; the variation model's per-pixel σ treats cross-sample texture misregistration as "process variation," and the golden sample itself is flooded.

These pitfalls shaped this chapter's experimental methodology: the threshold sweep and polarity statistics all use a **hand-written** $\mu \pm \max(a, k\sigma)$ **model** (32-bit float, fully deterministic), cross-validated against the SDK `CompareModel` output — contamination 9926 vs 10052 for 002, foreign object 3666 vs 3824 for 001, the total defect-pixel counts agreeing, confirming the two implement the same mathematics. This continues the golden-experiment methodology of Chapter 5: what the documentation describes is a promise; what a controlled experiment with known behavior verifies is the truth — and when some kind of data invalidates one prerequisite of a commercial library, let your own hand-written mathematical reference serve as the arbiter. The complete project lives at `code/defect_detection/`, with the full measurement log preserved in the file headers.

Industry Case: Mistaking the "Golden Sample" for a Cure-All

A nonwoven filter-material inspection line copied print-inspection experience and tried to do surface defect detection with a registered variation model: it collected twenty "good" images to build the model, formed the band from per-pixel mean and variance, and went live with a false-alarm rate over 30%, nearly every frame flooded with small regions. Investigation found the root cause lay not in the threshold, not in the lighting, but in **the fiber texture of the filter material being random from frame to frame and not lining up at the pixel level after registration** — the variation model treated texture misregistration as process variation, and the tolerance band was stretched until it

either false-alarmed everywhere or let real defects through. The final solution abandoned cross-sample registration and switched to **single-image self-reference**: each image is compared against its own large-window smooth background, with the band width set by the robust scale of that image's texture. The lesson: **the variation model has an invisible threshold — pixel-wise repeatability after registration**. Print, labeling, and screen printing satisfy it; fiber, fabric, frosted, and brushed surfaces do not. Before going live, do a "register two good parts and subtract" health check first: if the difference image is covered in texture, stop counting on a registered golden sample and switch to a self-reference or texture-statistics route.

26.6 Summary

- **The paradigm of defect detection is definition by inversion**: defect morphologies cannot be enumerated; the operational definition is "a deviation from normal" — first define normal, then quantify the deviation. Normal can come from registered good parts (the variation model), geometric rules (edge/contour defects), or within-image statistics (self-reference).
- **The registered variation model has an invisible threshold: pixel-wise repeatability**. Real fibrous filter disks do not satisfy it — register 5 disks and train, and even the golden sample itself is flooded with 1094 texture false-alarm regions. Print-like surfaces are its home turf; fiber/fabric/frosted surfaces are not.
- **The way out is a single-image self-reference band** $|I - \mu| > \max(a, k\sigma)$: μ taken as the image's own large-window smooth background, σ as the robust scale of the golden sample's residual (8.69). It cleanly exposes the contamination hole (assigned to background by Otsu, filled bright by the smooth background) as a large negative-residual defect — the contamination blob of 002 at 9738 pixels, contrast -101 gray levels.
- **Tune thresholds by squeezing from both ends**: acceptable blemishes press false alarms to give the lower

bound (golden-sample scratches 13384 at the tight setting → 236 at the loose), known must-catch real defects verify detection to give the upper bound (contamination pierces through all thresholds). On real samples the line between "defect" and "blemish" is decided by the acceptance criterion, not by physics.

- **Every prerequisite of a commercial library must pass a golden experiment**: the SDK `CompareModel` was repurposed as a self-reference detection engine and only after cross-validation against the hand-written band (9926 vs 10052) was confirmed to be the same mathematics; testing also forced out new pitfalls — `maxLength 100000`, circular ROI rejected — documentation is a promise; the experiment is the behavior.

And so the inspection part closes: blob analysis gave us "segment and measure," contour analysis gave us "point-by-point reconciliation," shape features gave us "a language for shape," and this chapter distills them all into the paradigm of "define normal, detect deviation" — and uses a set of real filter-disk samples to prove that when the paradigm meets the ground, the first thing that matters is asking the right question: where does "normal" come from? A map of methods for textured-surface defect detection can be found in Xie's review (Xie 2008), which systematically organizes the statistical, structural, filter-based, and model-based routes around texture analysis; the canonical source for texture-feature statistics remains Haralick et al.'s GLCM paper (Haralick, Shanmugam, and Dinstein 1973); and for the frequency-domain (Fourier) route not pursued in this chapter, Tsai and Hsieh's defect detection for directional textures is a clean exemplar (D.-M. Tsai and Hsieh 1999). For a more systematic engineering treatment of industrial defect detection (including textured surfaces and extensions of the variation model), see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

Part VII

Recognition

This part covers techniques for reading information from images and classifying it: 1D and 2D barcode recognition, optical character recognition (OCR), and classical classifiers that remain practical in industrial settings.

27 1D and 2D Barcode Recognition

Up to this point, every algorithm in this book has answered questions like "where is the target," "what are its dimensions," "is there a defect." The recognition part of the book, which begins with this chapter, answers a different question: **what is written in the image** — turning pixels back into data. The most mature, largest-scale "image to data" application on the factory floor is the barcode: every workpiece, every circuit board, every box of material carries a printed or marked code, every station on the line reads it once, and the entire supply chain's traceability system is built on those reads. Reading codes sounds easy — everyone scans codes with a phone — but codes on a production line are battered in turn by ink spread, lens defocus, low resolution, and smudging, and a single **misread** causes a part-mixing incident far more expensive than a failure to read. This chapter uses one real multi-symbology 1D barcode test card (Figure 27.1) to thread together four sets of experiments — 1D decoding, robustness boundaries under degradation, 2D error-correction comparison, and print-quality grading — and closes with a "hand-written encoder + SDK decoder" round-trip experiment.

27.1 1D Barcode Principles

The encoding idea behind a 1D barcode is remarkably plain: carry the data in a **sequence of bar and space widths**. The narrowest bar/space width is called the **module**; every other element's width is an integer multiple of it. A character consists of a fixed number of bars and spaces, and the wide/narrow combination is that character's code. The information unfolds along one direction only — every row in the vertical direction

Figure 27.1: Real multi-symbology 1D barcode test card `sample/barcodes.jpg` (1536×878 grayscale): a standard card carrying 8 symbologies / 9 symbols — top row: CODE39 and CODE128 both encoding "SCIMV", EAN-8 encoding 12345670; middle row: EAN-13 encoding 1234567890128, UPC-A encoding 185020919621, UPC-E encoding 01234565; bottom row: CODE93 encoding "Sci-Code93", two ITF (Interleaved 2 of 5) both encoding 123456789013. The red boxes and red labels are SciVision `FindBarcode` detections overlaid (symbology name + decoded string).

carries the same content. This is both the reason a single scan line suffices for decoding and the root of its robustness weakness; Section 27.2 will make that point painfully concrete.

The common **symbologies** divide the work among themselves: **CODE39** encodes one character with 9 elements (5 bars and 4 spaces, 3 of them wide), supports digits and uppercase letters, has a simple structure and variable length, and is extremely common on industrial labels — but beware that its check character is **optional**; the standard does not mandate it. **CODE128** is denser, supports the full ASCII set, and **mandates** a check character. **EAN-13/EAN-8/UPC** are retail product codes: a fixed number of digits with the last as a check digit. **CODE93** is a higher-density refinement of CODE39. **ITF (Interleaved 2 of 5)** is a digits-only **continuous** symbology — both bars and spaces carry information with no inter-character gaps, giving high density but, again, an **optional** check digit. The test card in Figure 27.1 places all 8 of these symbologies side by side, which is exactly what we need to compare how their constitutions differ under degradation.

Scan-line decoding proceeds in four steps. First, **locate the code** in the image — a barcode region has strong single-direction texture with highly consistent gradient orientations, which allows fast localization of candidate regions and estimation of their orientation. Then lay one or more **scan lines** along the barcode direction and extract a gray-level profile. Run edge detection on the profile to convert the alternating bars and spaces into a **width sequence**. Finally, look the width sequence up in the symbology's codeword table to recover the characters, and verify the start/stop characters (`*` for CODE39) and the check character (if present).

Each CODE39 and ITF element has some "self-checking" property — a single wide/narrow misjudgment usually lands on an invalid codeword and is discarded; but the string as a whole has no mandatory check character, so character-level truncation or substitution goes undetected. ITF is also continuous and the densest of the lot, and this hazard will be confirmed by its repeated misreads in Section 27.2.

Running the SDK's `FindBarcode` on Figure 27.1 (full project in `code/barcodes/`; since `barcodeType` has no "automatic" setting, the example tries each of the 8 candidate symbologies in turn and merges the results), all 9 symbols on the clean image **decode on the first attempt**, all oriented near 0° (the card lies horizontal), with pixels-per-module between **2.1 and 4.1 px/module** — CODE93 the narrowest (about 2.1), EAN-8 the widest (about 4.1). Turning on `checkQuality` yields ISO/IEC 15416 print-quality grades, and the spread is striking:

EAN-8 and UPC-E score a perfect **4.0**, CODE93 **3.0** (limited by **modulation**), CODE39 and EAN-13 **2.0**, CODE128 and UPC-A **1.0** (all limited by **decodability**), while the two **ITF symbols grade 0.0 (F)** — decodability is zero, the measured widths are pressed right against the codeword decision boundaries with essentially no margin. Decodability reflects the distance of the width measurements from the decision boundary; modulation reflects local contrast at the narrow elements. On one and the same card the best-printed code (EAN-8) and the worst (ITF) differ by a full 4 grades — which is exactly the value of the grading system: it measures not "did it read this time" but "how far from unreadable."

The value of the grading system is this: it does not answer "did it read this time," but "how far is this code from becoming unreadable." A grade-1.0 code that reads today will stop reading after a lens change or a bit of dirt; requiring a label grade of 2.5 or 3.0 at line acceptance is a far more reliable standard than "it scanned successfully during the site trial."

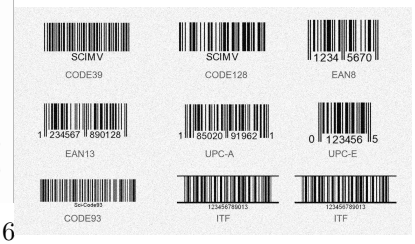
ISO/IEC 15416 grades 1D print quality from 4 (A) down to 0 (F): decode, contrast, modulation, defects, decodability, and other parameters are measured separately on multiple scan lines, each line is graded by its **worst** parameter, and the line grades are averaged. An overall grade 1.5 can usually be read reliably, but with little margin left.

27.2 Robustness Boundaries

A successful decode is only the starting point; what engineering really cares about is **where the boundary lies**. We subject all 9 symbols of Figure 27.1 to four kinds of progressive degradation — Gaussian blur, additive noise, downsampling, and occlusion — recording the survival and misread status of each symbology at every level (Figure 27.2 shows four representative levels).

Each of the four boundaries carries its own meaning, and together they confirm the clean-image grade ordering item by item — the lower-graded codes fall first. **Blur**: all 9 codes are fine at $\sigma = 1$, 6/9 survive at $\sigma = 2$, and the first to fall are exactly the narrowest-module CODE93 (about 2.1 px) and the two ITF (about 3.0 px); all are gone at $\sigma = 3$ — blur flattens the contrast of narrow bars and spaces, the narrowest codes die first, exactly matching the 15416 prophecy that ITF/CODE93 were already at the bottom. **Noise**: $\sigma = 10$ is harmless, only UPC-A drops out at $\sigma = 30$, and 6/9 still survive

(a) Gaussian blur $\sigma = 2$: 6 of the 9 codes still decode; CODE93 and both ITF fail, and one **ITF is misread as 120000000013**



(b) Noise $\sigma = 30$: only UPC-A fails, the other 8 codes are fine (all 9 fine at $\sigma = 10$)

(c) Scaling $\times 0.35$: only CODE39/CODE128 ("SCIMV") and EAN-8 remain readable; the rest fail below about 1.5 px/module

(d) A full-height white stripe occluding 10% of each code's width: **all 9 codes fail** (red boxes mark the stripe positions)

Figure 27.2: Four representative levels of the 1D robustness experiment (red overlays mark detections / occlusion stripes). Complete boundaries: blur — all 9 fine at $\sigma = 1$, 6/9 survive at $\sigma = 2$, all gone at $\sigma = 3$; noise — all 9 fine at $\sigma = 10$, 8/9 at $\sigma = 30$, 6/9 at $\sigma = 50$; scaling — 6/9 at $\times 0.5$, 3/9 at $\times 0.35$, all gone at $\times 0.25$; full-height white-stripe occlusion — 6/9 at 2%, 2/9 at 5%, all gone at 10%.

at $\sigma = 50$ — multiple scan lines provide a natural averaging effect against random noise, with considerable margin. **Resolution**: 6/9 survive at $\times 0.5$, and at $\times 0.35$ only the 3 widest, steadiest codes remain (the two SCIMV and EAN-8, original ppm about 3.7–4.1, 1.3–1.4 px/module after scaling), with all gone at $\times 0.25$ — 1D decoding only needs to resolve bar/space widths along a scan line, so its resolution requirement is far more lenient than intuition suggests, about 1.5 px/module being the practical floor here, and the narrowest CODE93 dies first the moment we downsample (for lens and resolution selection see Chapter 3).

The line most worth reading character by character is not which code fails to read, but which one **reads wrong**. At blur $\sigma = 2$ one ITF does not report failure — it **decodes as 120000000013** (true value 123456789013); at 2% and 5% occlusion the other ITF is **misread as 123400789013**; and at scaling $\times 0.5$ a nonexistent EAN-8 even appears, misread as 45252921. For a traceability system this is far more dangerous than a no-read: a no-read triggers a manual rescan, but **a misread that goes unnoticed** lets one workpiece travel the entire line wearing someone else's identity. The root cause is the foreshadowing planted earlier — the check digit of ITF (and CODE39) is optional and absent on this card; ITF is also a **continuous** symbology (both bars and spaces encode, with no inter-character gaps), so local damage easily collapses a segment into another valid even-length digit string that the symbology layer cannot notice, which is also why it grades F for decodability even on the clean image. There are two engineering countermeasures: either choose a symbology with a mandatory check character (**CODE128 never once misreads in this experiment**), or enable the optional check digit for ITF/CODE39 and turn on `codeCheck` verification at the decoder; beyond that, validating the content format (digit count, prefix) at the application layer is always worthwhile.

Occlusion best separates the codes by constitution: a white stripe running the full height of the barcode, widened step by step as a percentage of each code's width — at 2%, 6 of the 9 still read (the more forgiving CODE39/CODE128/EAN family survive, while the densest CODE93 and ITF have already fallen); at 5% only the two SCIMV remain; at 10% all fail. But

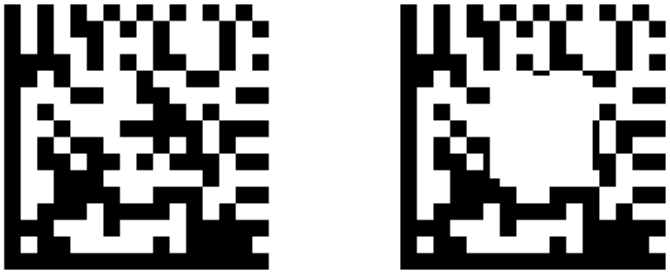
"can be read" does **not** equal "is read correctly": the two ITF misreads above occur exactly at the 2% and 5% occlusion levels. The reason is structural: a 1D barcode's redundancy exists only in the **vertical** direction — the same content is repeated over the entire bar height, and any single horizontal scan line can decode it, so smudges from above or below are not fatal; but horizontally each character is encoded exactly once, with no error correction at all, and a full-height vertical occlusion that wipes out one character wipes out its only copy. To survive horizontal damage, the redundancy must be encoded into the data itself — which is precisely the design starting point of 2D codes.

27.3 2D Codes and Error Correction

DataMatrix (ECC200) is the most common 2D code on industrial parts. Figure 27.3a is a 16×16 symbol: the entire left column and bottom row form the solid **L-shaped finder pattern**, providing the position and orientation reference; the top row and right column form the alternating black-and-white **clock track**, calibrating the row/column coordinates of the module grid; the remaining 14×14 **data region** places the codewords in the zigzag order prescribed by ISO/IEC 16022, with each codeword's 8 bits arranged as a "Utah-shaped" module group.

The robustness core of 2D codes is **Reed-Solomon error correction**. Intuitively it can be understood like this: treat the 12 data codewords as the coefficients of a polynomial and compute 12 additional **check codewords** over GF(256) to append after them — the 24 codewords are thereby bound by strong algebraic constraints, and when a few codewords are destroyed, the decoder can work backwards from the constraints to determine "which ones are wrong, and what their original values were." The 12 check codewords can correct about **6 erroneous codewords** at unknown positions — in this example only half of the 24 codewords are data: **half the symbol area is spent on redundancy**.

The Reed-Solomon error-correction budget: n_{ecc} check codewords can correct t erroneous codewords at unknown positions, subject to $2t \le n_{\text{ecc}}$. Here $n_{\text{ecc}} = 12$, so $t \le 6$: with 6 of the 24 codewords destroyed (25%), full recovery is still possible.



(a) A 16×16 symbol generated by the hand-written ECC200 encoder, content "MVBOOK CH27"

(b) A white patch occluding 15% of the symbol area: beyond the error-correction budget, decoding fails (still correct at 10%)

Figure 27.3: DataMatrix symbol structure and the occlusion experiment. (a) Left/bottom: L-shaped finder pattern; top/right: alternating clock track; interior: data region. (b) The off-center white patch avoids the finder pattern and damages only the data region; 10% of the area can be absorbed by Reed-Solomon error correction, 15% fails.

The occlusion experiment puts this "area for robustness" trade on the table: covering the symbol area with a white patch (placed off-center, avoiding the finder pattern), **decoding is still correct at 10% and fails at 15%** (Figure 27.3b). Compared with the 1D codes — the most fragile ITF/CODE93 already fall at 2% full-height occlusion and all 1D codes are gone by 10%, with misreads before they die — the 2D code not only tolerates more, but crucially **does not silently read wrong**: when the error-correction budget is exceeded, `ReadDM` simply returns failure (rc=122703001) rather than a wrong value. Moreover, DataMatrix decodes correctly under blur $\sigma = 1, 2, 3$ and noise $\sigma = 30$ — levels at which half the 1D codes had already been wiped out (to be fair, the DM here is about **10.3 px/module**, enjoying a resolution advantage over the card's 2–4 px/module 1D codes, but the occlusion comparison is unaffected by this). It must be stressed that error correction is not free: for the same amount of data, a symbol with 50% redundancy needs twice the area; what the 2D code buys with area is the **ability to survive horizontal damage** — something a 1D code cannot buy at any area.

A word on the **QR code**: it belongs to the same "matrix code + Reed-Solomon" family as DataMatrix, the main difference being the localization scheme — QR uses three nested-square finder patterns at the corners, while DataMatrix uses the solid L-shaped edges; the former favors fast localization of large symbols at a distance, the latter is more compact for small-size direct part marking (DPM). The SDK provides a `ReadQR` interface, but this chapter's test image contains no QR symbol, so no QR experiments were run; the principled conclusions carry over by analogy.

27.4 Round-Trip Verification Methodology

Where did the previous section's DataMatrix image come from? There is a methodological question here worth unfolding: the SciVision SDK **has only decoding APIs and no generation API whatsoever**, and the testImage directory contains no 2D code images. To run the 2D experiments, we **hand-wrote an ECC200 encoder** (about 150 lines) in

the example project: implement Reed-Solomon encoding over GF(256) (primitive polynomial 301; generator polynomial roots $\alpha^1 ... \alpha^{12}$), encode the text in ASCII mode (digit pairs compressed into one codeword, 253-state algorithm padding), then place the 12+12 codewords into the 16×16 symbol using the standard placement algorithm of ISO/IEC 16022 Annex F, and render the image with the finder pattern and clock track added.

The way to verify it is the **round trip**: hand the encoder's image to the SDK's `ReadDM` for decoding and compare the returned string character by character against the original "MV-BOOK CH27". Experimental result: decoding succeeds and the content matches; the symbol is 16×16 at about 10.3 px/module, with an ISO/IEC 15415 overall grade of **4.0** — a perfect score, exactly what an ideally rendered symbol should get. This round trip tests both ends at once: if the encoder places even one bit in the wrong position, decoding fails or the error-correction margin drops; and the SDK decoder reading back a perfect grade also proves that its grid localization and quality measurement work correctly. This is a reprise of the "gold-standard experiment" idea from Chapter 5 — **test the entire chain with an input of known ground truth** — and here the ground truth was encoded by our own hands, every bit traceable. It also demonstrates another use of the 15415/15416 grades: with a perfect-grade symbol as the baseline, a field symbol's grade shortfall can be attributed parameter by parameter to the printing and imaging stages (for contrast problems caused by illumination, see Chapter 4).

27.5 SciVision Implementation

1D decoding is done by `SCIMV::SciSvBarCode::FindBarcode`:

```
SCIMV::SciSvBarCode bc;
Sci1DCodeArray codes; SciROIArray rects, bad;   // pass concrete objects for all three outputs
long rc = bc.FindBarcode(img, roi, SCI_CODETYPE_CODE39,
                         1,      // polarity: dark bars on light background
                         0,      // sampleRate: automatic sampling density
                         2,      // contrastLevel: medium contrast
```

```
                          4,      // codeNumber: expected count (headroom for per-symbology scan
                          10000,  // timeOut: timeout (ms)
                          0,      // codeCheck: check-character verification switch
                          0,      // decodeDirection: decode in both directions
                          false,  // sendStartEnd: do not output start/stop characters
                          true,   // checkQuality: output ISO 15416 grades
                          &codes, &rects, &bad);
// Sci1DCode: codeType / codeString / orientation
//            / pixelPerModule / qualityLevel_15416
```

Three parameters deserve special mention. With `codeCheck` on, the check character is verified per the symbology's rules — for CODE39 this requires the printing side to include the optional modulo-43 check character, but it is the symbology-level defense against the misread incident of Section 27.2. With `checkQuality` on, the per-parameter 15416 grades are output for each code — keep it on during line debugging and acceptance, and turn it off in stable production to save time. `timeOut` is the decoding timeout — stations with tight cycle times must set it, or a single poor-quality image can drag down the whole cycle.

DataMatrix decoding uses `SCIMV::SciSvMatrixCode::ReadDM`:

```
SCIMV::SciSvMatrixCode mc;
SciDMCodeArray ret; SciVarArray strs; SciROIArray cand;
long rc = mc.ReadDM(img, roi, 1 /*polarity: dark code on light background*/, -1 /*dmStyle: auto
                    6, 20,  // minSize/maxSize: search range for symbol rows/columns
                    30,     // minContrast: minimum contrast
                    0,      // mirror: detect mirroring automatically
                    1, 10000, 0,            // codeNum / timeOut / model
                    true,   // checkQuality: output ISO 15415 grades
                    false, false, false, false, false, false,
                    1, 1,   // decode and output strings both as UTF-8
                    &ret, &strs, &cand);
// SciDMCode: codeString / symbolRows / symbolCols
//            / moduleSize / orientation / qualityLevel_15415
```

Two engineering pitfalls. First, **the symbology must be specified**: `FindBarcode`'s `barcodeType` has no "automatic"

setting, and asking for a symbology that is not in the image fails (returns a non-zero error code); when the symbology is unknown, the only option is to loop over a candidate list as this chapter's example does (here it calls `FindBarcode` once for each of the 8 1D symbologies and merges/dedupes) — but a production project should pin the symbology down at the design stage, as part of the specification agreed with the label supplier. Second, header files such as `SciSvBarCode.h` unconditionally define their export macro as dllexport; placing their `#include` **after** the other dllimport-style headers makes linking work normally.

Industry Case: Read Rate from 98% to 99.9%

The DataMatrix read rate on an assembly line was stuck at 98% — which sounds high, but it means 20 manual rescans per thousand parts, unacceptable for both cycle time and labor cost. The project team first tweaked the decoding parameters repeatedly with little effect, then measured the labels with a verifier: modulation was only grade C — the label stock absorbed ink, causing ink spread and collapsing the contrast of the narrow elements; auditing the optical chain at the same time revealed a module width of only about 1.8 px/module, right at the decoding floor. Two corrective actions: switch to high-contrast laminated label stock, bringing the grade back to A/B; and increase the lens magnification to reach 3 px/module. The read rate rose to 99.9%, and the decoding parameters ended up at their defaults. The lesson: **for read-rate problems, check the print-quality grade and the pixels per module first, then go tune algorithm parameters** — when the grade is failing, tuning parameters is just gambling at the boundary.

27.6 Summary

- 1D barcodes encode in bar/space widths and decode along scan lines; they have a full bar height of redundancy vertically but no error correction at all horizontally: on this 8-symbology card the densest ITF/CODE93 already fall at **2%** full-height occlusion and all 1D codes are gone by **10%**; the resolution floor is about 1.5 px/module (only

the 3 widest codes survive $\times 0.35$). The survival order under degradation confirms the clean-image 15416 grade ordering item by item.

- **A misread is more dangerous than a no-read**: on this card ITF (optional check digit, continuous symbology) misreads several times under blur $\sigma = 2$ and 2%/5% occlusion (e.g. 120000000013, 123400789013) and grades F for decodability even on the clean image — choose a symbology with a mandatory check character (CODE128 never misreads here), turn on `codeCheck`, and validate the format at the application layer: at least one of these three defenses must be in place.
- 2D codes use Reed-Solomon error correction to **trade area for robustness**: the 16×16 DataMatrix's 12 check codewords can correct about 6 erroneous codewords, decoding remains correct with 10% of the area occluded, and beyond budget it returns failure rather than a misread — a horizontal-damage tolerance far above the 1D codes.
- When the SDK only decodes and never encodes, the **hand-written encoder + SDK decoder round trip** both fills the experimental gap and constitutes a gold-standard verification of both ends (here the round trip scored a perfect ISO 15415 grade of 4.0).
- The ISO/IEC 15416/15415 print-quality grades answer "how far from unreadable": accept on grades rather than on trial reads, and for read-rate problems check the grade and the pixels per module first.

For a more systematic treatment of barcode recognition and decoding algorithms in industrial vision, see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018). The print-quality grading systems cited repeatedly in this chapter come from two core standards: the per-parameter grading method for 1D codes is specified by ISO/IEC 15416 (International Organization for Standardization 2016), and the symbol-quality grading for 2D codes (including Data Matrix and QR) by ISO/IEC 15415 (International Organization for Standardization 2011c). The encoding, error correction, and reference decode algorithm of each symbology have their own dedicated standards: the codeword placement and Reed-Solomon rules of Data Matrix

(ECC200) are in ISO/IEC 16022 (International Organization for Standardization 2006), and QR Code in ISO/IEC 18004 (International Organization for Standardization 2015) — when agreeing on a specification with a label supplier or reproducing this chapter's experiments, the original text of these standards should be the authority.

28 Optical Character Recognition

When OCR (Optical Character Recognition) comes up, most people think of scanners and document digitization: thousands of character classes, endlessly varied fonts and layouts. Industrial OCR is an entirely different business. What the production line has to read is the production date an inkjet printer sprays onto a bottle cap, the serial number a laser engraves into a metal part, the lot number a dot-peen marker punches into a nameplate — the character set is often just a few dozen symbols (digits plus a handful of capital letters), and the font is dictated by the marking equipment and completely fixed, but the imaging conditions are far harsher than a scanned document: curved reflective surfaces, ink of uneven density, strokes broken by clogged nozzles. Under this combination of "small character set, fixed font, hostile surface," the classical **trainable font** approach remains the workhorse to this day: train a small classifier on real characters collected on site, and its robustness to this particular font far exceeds any general-purpose document OCR. In this chapter we walk this route end to end with a custom dot-matrix font — training, recognition, stress testing, until we push it to failure with our own hands. Figure 28.1 is our "font": 16 characters, 0–9 and A–F, each 20×28 pixels, ink gray level 50 on a background of 200, deliberately including confusable pairs such as 0/D and 8/B.

28.1 The Classical OCR Pipeline

Classical OCR decomposes "reading characters" into four steps, each built on the foundations of earlier chapters.

Figure 28.1: Training row of the custom dot-matrix font: 16 characters, 0–9 and A–F, each 20×28 px, ink gray 50 on background 200. The character set deliberately includes confusable pairs such as 0/D and 8/B, planting the seed for the confusion experiments to come.

1. **Binarization**: separate the character strokes from the background to obtain the foreground region. This is the thresholding of Chapter 7 — character-to-background contrast is usually high, so a fixed or automatic threshold is up to the job; the key is choosing the right polarity (dark characters or bright ones).
2. **Character segmentation**: cut the foreground region into individual character units. Its core is the connected-component analysis of Chapter 23, overlaid with the structural priors of text: characters line up in rows, spacing within a row is roughly uniform, and character width, height, and area fall within known ranges.
3. **Feature extraction / normalization**: rescale each character crop to a uniform size and extract gray-level or shape features.
4. **Classification**: map the features to a character label with a classifier — exactly the subject of Chapter 29, except that the classes change from "good/defective" to 16 characters.

Note one structural fact: these four steps form a **one-way pipeline**, where each step's output is the next step's input unit. However clever the classifier, it can only answer for the patch of image that segmentation hands it — this seemingly trivial observation will reveal its full weight in Section 28.5.

Deep-learning OCR (the SDK has a separate DLOCR module) performs detection and recognition end to end, well suited to scenes with variable fonts and complex backgrounds; but for inkjet-code reading with a fixed font and a small character set, the classical route needs few training samples, runs fast, and fails in explainable ways — it remains the lighter and more controllable choice. This chapter does not cover DLOCR.

28.2 Training the Font

SciVision's classical OCR workflow is a three-act play: segment — label — train. We first render one clean training row of the 16 characters (Figure 28.1), binarize it with `GetCharRegion` to obtain the character foreground region, then use `SegCharacters` to cut out 16 character crops following the row/column structure. Since the character order of the training row is known (it is simply "0123456789ABCDEF"), sorting the crops left to right by bounding-box center makes the labels line up automatically — `AddSamples` ingests the 16 binary crops and 16 label strings in one call.

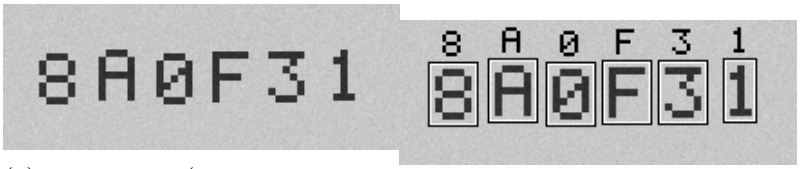
A font trained on a single clean row would be too "delicate": the classifier has never seen noise or positional offset, and could easily fall over on real inkjet print. So we apply light **data augmentation**: beyond the clean master, we render 3 more degraded rows — adding Gaussian noise of standard deviation $\sigma = 5$ and randomly jittering each character by ± 1 pixel — then segment, label, and `AddSamples` them the same way. Four rows total $16 \times 4 = 64$ samples over 16 classes; finally we call `Train(0)` to train the default classifier (which takes binary crops as input). After training, a self-check on the clean training row recognizes all 16 characters correctly.

That it works with only 4 samples per class sounds like a violation of machine-learning intuition, but the preconditions are extremely strict: **the font is completely fixed** (within-class variation comes only from noise and tiny displacements, with no glyph differences) and **segmentation is reliable** (what the classifier receives is always one complete, centered character). These two preconditions are precisely the watershed between industrial OCR and document OCR — the former trades control of the scene for small sample counts, the latter throws massive datasets at uncontrolled diversity. The moment a precondition breaks (the marking equipment swaps its character dies, characters touch and merge), the small-sample font fails instantly — which is exactly the subject of the experiments in the next two sections.

28.3 Recognition and Confidence

The font is trained; let us read a string it has never seen. The test string "8A0F31" is rendered with $\sigma = 5$ noise and a random jitter of ± 1 px per character (the upper part of Figure 28.2 shows the annotated result), pushed through the same binarize-and-segment pipeline to extract 6 characters, and handed to `Recognize`. The result is **all correct**, but what deserves a closer look is the score (0–100) of each character:

Position	Truth	Recognized	Score
0	8	8	100.00
1	A	A	100.00
2	0	0	100.00
3	F	F	94.95
4	3	3	100.00
5	1	1	87.16



(a) Test string ($\sigma = 5$, ± 1 px jitter) (b) Annotated recognition result

Figure 28.2: Recognition of the test string "8A0F31". (a) Input image; (b) each segmented character is framed with a double-line box, with the recognized glyph annotated above it — all 6 characters are correct, but the scores range from 100 down to 87.16.

Why does '1' score only 87.16? It is the narrowest glyph in the character set (just 11 px wide), with the fewest stroke pixels, so after normalization and rescaling, the noise and the 1 px jitter affect it the most in relative terms; the 94.95 of 'F' likewise comes from noise eroding the open ends of its strokes. The value of the score lies not in "how much was right" but in being **the basis for rejection and review**: the `minScore` parameter of `Recognize` draws the acceptance threshold, and

a character below it does not output a guessed value but the placeholder specified by `replaceChar` (we pass "?").

The engineering semantics of this parameter pair deserve special emphasis: **better to output "?" and trigger manual review or a rescan than to silently emit a low-confidence guess**. A misread production date that reaches the market costs far more than one rescan on the line — this is the same principle that Chapter 26 hammers on ("silent failure is the most dangerous failure"), projected onto OCR. The gap between 87.16 and 100 caused no error today, but it tells you something: '1' is the character standing closest to the cliff edge in this font, and where to set `minScore` — and how much margin to leave — should be read out of exactly this kind of score distribution.

28.4 Confusion and Limits

The font handles $\sigma = 5$ with ease — so where is its limit? We designed a confusion stress test: the test string "0D8B" simultaneously contains two pairs of highly similar glyphs — 0 and D (both closed loops, differing only in corner roundness) and 8 and B (differing only in whether the left side pinches at the waist). The noise is stepped up from $\sigma = 10$ to 50, with 10 independent render-and-recognize trials per level.

The result is unexpectedly "steep": the $\sigma = 10$, 20, and 30 levels are **all 100% correct** — at $\sigma = 30$ the image is already a blizzard of noise, yet the font stands rock solid. The first failure appears at $\sigma = 40$: per-character accuracy drops to 90%, and the failure refuses to follow the script — it is not the anticipated 0/D or 8/B swap, but **D recognized as C**. Figure 28.3 is the scene of that first failure: the $\sigma = 40$ noise happened to "eat away" the vertical stroke on the right side of the D, opening the closed loop — and in the classifier's eyes it became a C opening to the right. The failure mode at $\sigma = 50$ is the same. Across the entire experiment of 50 trials there was **not a single segmentation failure** — segmentation is inherently more robust than classification, because it only needs

"there is a blob of foreground here," not to tell what that blob is.

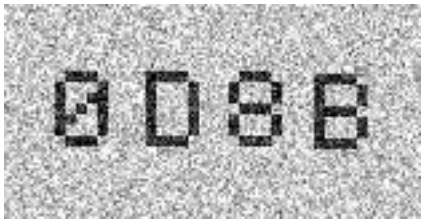


Figure 28.3: First recognition failure at the $\sigma = 40$ level: the D in "0D8B" is recognized as C — the noise eroded precisely the vertical stroke on the right side of the D, turning the closed loop into an open glyph. The failure mode is not the anticipated 0/D swap.

This experiment yields two actionable conclusions. First, classical OCR on a fixed-font, small character set has a large robustness margin (it saw only $\sigma = 5$ in training and withstood $\sigma = 30$) — no need to worry about normal production-line fluctuations. Second, evaluating a font demands **measured confusion analysis**: crank the noise up until it fails and record the actual confusion pairs — they are often not the ones you predicted — and both the rejection threshold and the character-set design (e.g., serial-number schemes that avoid confusable characters) should be based on measured confusion, not intuition.

We carefully designed two pairs of "theoretically confusable" characters, 0/D and 8/B, yet the real first failure was D→C. The lesson: **failure modes do not follow intuition**. The confusion matrix must be measured, not deduced from human judgments of glyph similarity — what looks alike to a human, the classifier may not confuse; what looks unalike, one stroke of noise can make neighbors.

28.5 Segmentation: OCR's Achilles' Heel

Segmentation failed zero times in the confusion experiment because the character spacing was generous. Now remove that precondition: squeeze the character spacing of "8A0F31" to 0, so the strokes of adjacent characters touch directly — simulating the runaway character pitch or ink bleed common in inkjet coding. The result is in Figure 28.4: the strokes of the first 5 characters "8A0F3" merge into one large connected component about 100 px wide, and only the '1' is spared because its glyph leaves empty space on its right anyway. The segmenter dutifully reports: **2 "characters" extracted**. The recognition result is "?1" — that 100 px behemoth matches no glyph

in the font within the 0.5–2× scale search range and is rejected by the SDK as "?".

Figure 28.4: Test string with character spacing squeezed to 0: the strokes of the first 5 characters "8A0F3" touch and merge into a single connected component, and only the '1' stays separate. The segmenter extracts 2 "characters", and the recognition result is "?1".

Take a moment to appreciate how this failure differs in kind from the noise failure of the previous section. Noise turning D into C is an error at the **classification layer** — one character is wrong, the rest proceed as usual, and the score even raises a warning. A segmentation failure is an error at the **structural layer**: 6 characters became 2 input units, and no downstream classifier, however strong, can undo that — its job is to answer "which character is this unit," and "this unit is not a character at all" lies outside its question space. This is the Achilles' heel of classical OCR: **segmentation decides success or failure, and segmentation errors are unrecoverable**. In the research history of document OCR, touching character segmentation was one of the most stubborn problems, and the rise of end-to-end deep learning was in large part precisely a way to bypass explicit segmentation.

The industrial countermeasure is not to conquer this problem but to keep it from happening:

- **Regulate character spacing at the source**: the character spacing of marking equipment is configurable; write it into the process specification and leave segmentation some margin — this is cheaper than any algorithmic remedy;
- **Tie segmentation parameters to character dimensions**: the width, height, and area bounds of

`SegCharacters` should be set from the actual character die dimensions, so that a 100 px wide merged blob falls squarely outside the legal width range;

- **Alarm on touching detection**: expecting 6 characters but extracting 2 is itself a strong anomaly signal — alarm and re-inspect whenever the character count disagrees, and never release a mutilated result as normal output.

28.6 SciVision Implementation

All experiments in this chapter are carried out by a single class, `SCIMV::SciSvOCR`, whose four-step API maps one-to-one onto the pipeline of Section 28.1:

```
SCIMV::SciSvOCR ocr;
SciRegion region;
// 1) Binarize to extract character foreground: manual threshold [0,125] selects dark characte
//    Note: the measured semantics of polarity are the opposite of the docs -- dark characters
ocr.GetCharRegion(img, roi, /*thresholdMode*/0, /*polarity*/1,
                  /*minGray*/0, /*maxGray*/125, 0, 3, 3, 0, &region);

SciImageArray binImgs, greyImgs;  SciRegionArray regs;
SciROIArray lineRects, charRects; SciVarArray idx;
// 2) Character segmentation: 1 text line, 6 expected characters, width [8,120] height [14,56]
ocr.SegCharacters(region, roi, 1, 6, 10, 2, -10, 10, /*ignoreEdge*/0,
                  10, /*mergeMode*/1, 6, 2, 10, 0, 9999,
                  8, 120, 14, 56, -30, 30, 30, 9000, 0.0f, 5.0f,
                  &binImgs, &greyImgs, &regs, &lineRects, &charRects, &idx);

// 3) Training: feed each row's segmentation result with its labels, 4 rows = 64 samples, then
ocr.AddSamples(binImgs, labels);   // ×4 rows
ocr.Train(0);                      // 0 = default classifier, input is binary crops

// 4) Recognition: below minScore, output replaceChar ("?"); scores range 0-100
SciVar replaceChar("?"), result;  SciVarArray scores;
Sci2DVarArray resultAll, scoresAll;
ocr.Recognize(binImgArray, /*minScore*/0, 0.5f, 2.0f, 0.5f, 2.0f,
              replaceChar, &result, &scores, &resultAll, &scoresAll);
```

Among the key parameters, `thresholdMode=0` of `GetCharRegion` is a manual threshold, selecting dark strokes together with the gray-level interval `[0,125]`; the character width/height and area bounds of `SegCharacters` are both a noise filter and the anti-touching defense line of Section 28.5; the `Xscale`/`Yscale` intervals of `Recognize` permit a 0.5–2× scale search over each character during recognition.

In practice we hit three pitfalls that must be recorded honestly:

- **Under manual thresholding, the `polarity` semantics of `GetCharRegion` are the opposite of the documentation**. Per the docs, dark characters should pass 0, but in practice passing 0 yields the **background** connected component — the whole row of 16 characters becomes a single 538×78 "mega-character," and downstream segmentation collapses entirely. Dark characters must pass `polarity=1` together with the gray-level interval `[0,125]`.
- **`SegCharacters` with `ignoreEdge=1` spuriously raises error code 122701003**, even when no character touches an edge. Pass 0 to work around it.
- **The automatic width bounds of `AutoSegCharacters` (the automatic segmentation variant) drop narrow characters**. The width interval it estimates from the samples is [19,20], while '1' is actually only 11 px wide and gets filtered straight out — only 14 of the 16 characters survive. For fonts with large character-width variation, use `SegCharacters` with manual parameters and give the width bounds explicitly (this chapter uses [8,120]).

The complete runnable project is at `code/ocr/`, with a fixed random seed; every number is reproducible.

Industry Case: The Ghost 8 in Inkjet Date Codes

A bottle-cap date-code reading system at a food plant intermittently read the "3" in the date as "8". The investigation traced it to a slightly clogged print head: drifting ink droplets put a faint bridging stroke across the open left side of the "3", pushing the glyph toward "8". The key evidence was the score

curve — a normal "3" scored above 95, while during the fault it dropped to 60–70, **yet still above the `minScore=50` set at the time**, so the wrong result was released by the system as a valid read, exposed only by a customer complaint. The remediation came in three steps: raise `minScore` to 85; have any read below 95 trigger an automatic rescan with image retention; add print-head pressure and nozzle condition to the equipment inspection checklist, cutting off glyph drift at the source. The lesson: **the confidence threshold must be derived backward from the cost of a misread**, not set as leniently as possible — a threshold of 50 means the system defaults to "release even when in doubt," and for a date printed on food and facing the consumer, that default is unacceptable.

28.7 Summary

- **Industrial OCR document OCR**: small character set, fixed font, hostile surfaces. Trade control of the scene for small sample counts — 4 augmented samples per class suffice to train a usable font, but only if the font is fixed and segmentation is reliable; break a precondition and the font fails.
- **The classical pipeline = binarization → character segmentation → features → classification**, built respectively on thresholding, connected-component analysis, and classical classifiers; it is one-way, and every step's errors are paid for in full downstream.
- **Scores are the basis for rejection and review, not decoration**: "8A0F31" was all correct, yet the gap between 87.16 and 100 marked the character closest to the cliff. Derive `minScore` backward from the cost of a misread — better to output "?" than to guess silently.
- **Confusion must be measured**: in the -ladder experiment the font withstood $\sigma = 30$, six times its training noise, while the first failure at $\sigma = 40$ was the unanticipated D→C (noise opened the closed loop), not the carefully designed 0/D or 8/B — failure modes do not follow intuition.

- **Segmentation is OCR's Achilles' heel**: at zero spacing, 6 characters merged into 2 input units and recognition was reduced to "?1" — unrecoverable by any classifier however good. The countermeasures live on the engineering side: regulate spacing, tie segmentation parameters to character-die dimensions, and alarm whenever the character count disagrees.

For a systematic treatment of feature design for OCR classifiers and trainable fonts, see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018). For the broader arc of OCR research, the classic survey by Mori, Suen, and Yamamoto (Mori, Suen, and Yamamoto 1992) traces the field's history from template matching to structural analysis; for the feature-extraction stage this chapter dwells on, Trier, Jain, and Taxt provide a still-valuable survey of feature methods for segmented characters (Trier, Jain, and Taxt 1996); and to extend the scope to handwriting, the on-line and off-line recognition survey by Plamondon and Srihari is the recognized entry point (Plamondon and Srihari 2000). For 1D barcode and 2D code reading — OCR's siblings in the "reading printed information" family — see Chapter 27.

29 Classical Classifiers

Throughout the recognition part of this book we have, in fact, been doing classification all along — only in increasingly free forms. Barcode recognition is essentially "table lookup": the symbology pre-encodes the answer into black-and-white bars and spaces, and the algorithm merely decodes; OCR widens the scope to a few dozen character classes, but the character set is known and closed. This chapter faces the general classification problem itself: given a target, decide which of the predefined classes it belongs to — is that blemish on a tablet a black dot, a fiber, or a clump of debris? Problems like this have no encoding rules to lean on; statistical regularities can only be learned from samples. Before deep learning swept through the vision field, **k-nearest neighbors (KNN)**, the **multilayer perceptron (MLP)**, and the **support vector machine (SVM)** were the "big three" of pattern recognition; to this day, in industrial tasks with low feature dimensionality, small sample sizes, and a need for interpretability, they remain the kings of cost-effectiveness — training takes seconds, prediction takes microseconds, no GPU is required, and when a misclassification occurs you can trace the cause. This chapter threads a foreign-object sorting task through all three classifiers: distinguishing three kinds of dark targets on a bright background — round dots, elongated fibers, and irregular clusters — as shown in the first three rows of Figure 29.1; the "fat capsules" in the fourth row are deliberately manufactured borderline samples, which we save for making trouble later.

29.1 Feature Space and Separability

Classifiers do not consume images directly. The first step is to turn each image into a **feature vector**: after threshold segmentation yields the target region, the region features introduced

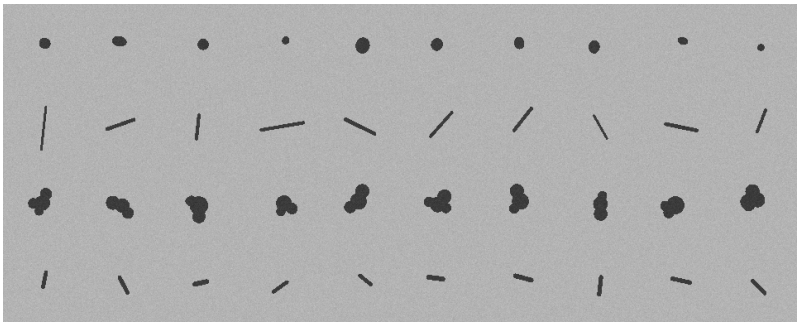


Figure 29.1: Sample images for the three-class task. The first three rows are dots, fibers, and clusters (the first 10 of 60 samples per class, 96×96 grayscale images with $\sigma = 5$ Gaussian noise added); the fourth row contains 10 "fat capsule" borderline samples deliberately placed between dots and fibers.

in Chapter 23 compress it into 4 numbers — area, anisometry, circularity, and compactness. Each sample thus becomes a point in a four-dimensional **feature space**, and "classification" becomes the geometric problem of carving up territory in this space.

What does real production foreign matter look like? Figure 29.2 is taken from one frame of the OPT Smart3 deep-learning sample project — a translucent cosmetic tube with several dark foreign particles scattered across its surface. We process it through exactly the same pipeline as above: place an ROI over the particle cluster, threshold-segment (the tube body is bright, the particles near-black), use blob analysis to filter out the printed halftone dots, and then extract the same 4-D region features for each particle. The 6 segmented particles and their real measured values are:

Particle	area	aniso	circ	comp	morphology
P1	1143	1.67	0.500	1.40	blocky chip (cluster-like)
P2	902	1.64	0.478	2.14	blocky chip (cluster-like)
P3	1180	2.29	0.358	2.21	blocky chip (cluster-like)
P4	562	1.14	0.753	1.09	compact dark spot (dot-like)
P5	294	1.33	0.571	1.20	compact dark spot (dot-like)
P6	322	3.28	0.280	1.73	elongated sliver (fiber-like)

These real measurements confirm that the synthetic task setup is not invented out of thin air: P4 and P5 are compact dark spots (circularity 0.57–0.75, anisometry near 1), corresponding to "dots"; P1–P3 are larger blocky chips with broken edges (circularity 0.36–0.50), corresponding to "clusters"; P6 is an elongated sliver (anisometry 3.28, circularity only 0.28), leaning toward "fiber." **Real foreign matter does spread along the same dot–fiber–cluster morphology continuum**, so the feature design of this chapter rests on real ground.

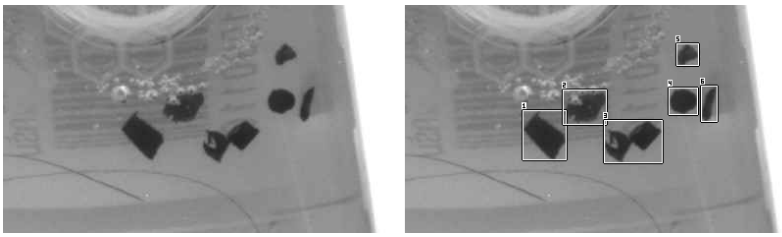


Figure 29.2: Feature extraction on real foreign matter (OPT Smart3 deep-learning sample image, 1280×1024 grayscale). Left: grayscale image of the particle cluster region on the translucent tube; right: the 6 particles segmented by threshold (bright tube, near-black particles) plus blob analysis, with bounding boxes numbered P1–P6 left-to-right by centroid abscissa. The real particles cover all three morphologies — compact dots (P4/P5), an elongated sliver (P6), and blocky clusters (P1–P3) — in one-to-one correspondence with the three synthetic classes.

But precisely because it is "real," the classifier training and evaluation cannot rest on it, for three reasons. First, a single frame holds only 6 particles and carries no per-particle ground-truth class labels, so there is no way to assemble a 40/20 train/test split per class. Second, the morphology spectrum of real foreign matter is continuous — P6's anisometry of 3.28 sits right at the lower edge of the "fat capsule" borderline interval (3.33–5.16) we synthesize later, leaving no clean gap between classes to compare the three classifiers' boundary behavior fairly. Third, a controlled synthetic set lets us dial separability, sample size, and boundary difficulty precisely — exactly the "test bench"

the experiments below require. **The division of labor in this chapter is therefore: the real sample image verifies that feature extraction and the morphology spectrum are genuine, while classifier training and the comparative experiments run on a controlled synthetic set that is labelable, splittable, and difficulty-adjustable** (the synthetic samples are shown in Figure 29.1).

Figure 29.3 projects this controlled synthetic dataset onto the anisometry × circularity plane. The layout of the three classes is plain at a glance: the fibers (crosses) hang alone in the lower right — anisometry 7.15–18.0, circularity a mere 0.07–0.15, neighbors to no one; the dots (filled disks) and clusters (hollow squares), by contrast, crowd together in the upper left — the dots' anisometry range of 1.00–1.48 almost coincides with the clusters' 1.09–2.17, and their circularity intervals (0.50–0.79 versus 0.41–0.68) overlap over a wide stretch. On these two dimensions alone, no classifier whatsoever could pull them apart. The rescue comes from the other two dimensions: dot area runs 54–340, cluster area 373–807 — one clean cut; compactness (0.90–1.09 versus 1.17–1.66) is just as cleanly divided. The gray diamonds are those 10 borderline capsules, with anisometry 3.33–5.16, perched squarely in the vacuum zone between dots and fibers.

This figure delivers the first — and possibly the most important — conclusion of this chapter: **separability depends first on feature selection, and only then on the classifier**. With the right features, the three classes sit far apart in four-dimensional space and even the plainest classifier gets everything right; with the wrong features — say, only the first two dimensions — the most ingenious model has nothing to work with. When troubleshooting poor classification results in engineering practice, you should first plot the feature scatter and check separability, rather than rushing to swap models and tune parameters.

The division of labor between feature engineering and end-to-end learning: deep networks hand "feature extraction" over to learning as well, at the cost of needing large amounts of labeled data; classical classifiers hand the features to domain knowledge (shape, texture, color statistics), leaving learning responsible only for drawing boundaries in a low-dimensional space. Industrial tasks often have few samples but well-understood mechanisms — landing squarely in the latter's comfort zone.

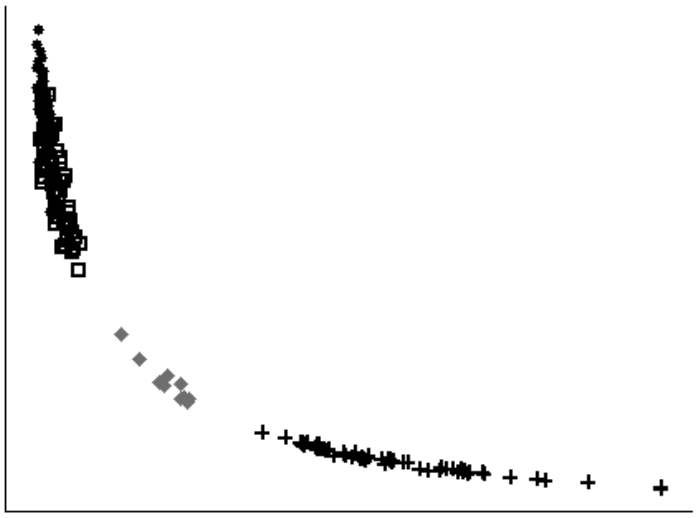


Figure 29.3: Feature-space scatter plot (horizontal axis anisometry, vertical axis circularity). Filled disks = dots, crosses = fibers, hollow squares = clusters, gray diamonds = borderline capsules. The fibers own the lower right; the dots and clusters overlap in this two-dimensional projection and can only be separated with the help of area and compactness.

29.2 The Geometry of Three Classifiers

All three classifiers learn a mapping from vectors to classes in feature space, $f : \mathbb{R}^4 \to \{\text{dot}, \text{fiber}, \text{cluster}\}$, but their "world-views" could hardly differ more.

KNN does not learn; it memorizes. The training phase merely stores all the samples (in practice building a kd-tree to speed up retrieval); at prediction time it finds the K training samples nearest to the query point and lets them vote. Its decisions are entirely **local**: the boundary is determined by the positions of a handful of samples, and if the samples carry noise, the boundary turns mottled and jagged along with them. Its strengths spring from the same source — zero training cost, instant effect when samples are added or removed, and every verdict can point to "which neighbors cast the votes." Since the features differ wildly in scale (area in the hundreds, circularity below 1), the data must be normalized before distance computation, or the area dimension alone will dominate the distance.

The MLP learns a smooth function. The input layer's 4 nodes correspond to the features, the hidden layer applies a nonlinear transformation, and the output layer's 3 nodes, after normalization, give a **soft score** for each class, with the maximum taken as the verdict. Training fits iteratively via backpropagation — it is the only one of the three that has to "learn slowly." Its decisions are **global**: all training samples jointly shape one continuous surface, individual noisy samples get averaged away, and the boundary is therefore smooth. The price is randomness in training, a hidden-layer size that must be specified by hand, and the fact that the soft scores are merely a byproduct of curve fitting, not calibrated probabilities — a point illustrated with a concrete example later.

The SVM cares only about the vicinity of the boundary. It seeks the hyperplane that separates two classes with the maximum **margin**; the hyperplane is determined solely by the few **support vectors** hugging the margin — no matter how many samples sit farther away, they play no part. Combined with the **kernel trick** — this chapter uses the radial basis function (RBF) kernel — it can draw a linear boundary

in an implicit high-dimensional space, which becomes a curved boundary back in the original space. The decision function takes the form

$$f(\mathbf{x}) = \operatorname{sign}\Big(\sum_i \alpha_i y_i \, k(\mathbf{x}_i, \mathbf{x}) + b\Big),$$

where the sum runs only over the support vectors. Multiclass classification is assembled from multiple two-class decisions.

The three worldviews projected onto a plane become Figure 29.4. We fix area and compactness at their training-set means (301.5 and 2.15) and query the three trained models point by point over a 72×72 grid on the anisometry × circularity plane. KNN (left) shows a mottled upper half — the fixed area value happens to fall between dots and clusters, this region is close to neither class's samples, the verdict flip-flops in a tug-of-war between neighbors, and isolated "enclaves" are scattered on the right; the MLP (center) produces a textbook-clean smooth boundary; the SVM's boundary (right) is stitched from several nearly straight arcs — each one the meeting line of the RBF influence zones of a few support vectors. The three panels classify the same data with near-identical accuracy, yet the shapes of the boundaries faithfully expose each model's **inductive bias**: local memory, global fitting, maximum margin.

The choice of K is KNN's only hyperparameter: $K = 1$ gives the most fragmented boundary and the highest noise sensitivity; as K grows the boundary smooths out, but small classes get "drowned" by large ones. This chapter uses $K = 3$ — enough for an actual vote (so a single noisy neighbor cannot decide alone), yet not so large as to flatten the local structure at a scale of 40 samples per class.

29.3 Experiment: Accuracy versus Sample Size

The 60 samples per class are split 40/20 into training and test sets, and we then cut the training set down to 5 per class, and even 2 per class, to examine the effect of sample size. The full results follow (the test set stays fixed at 3×20):

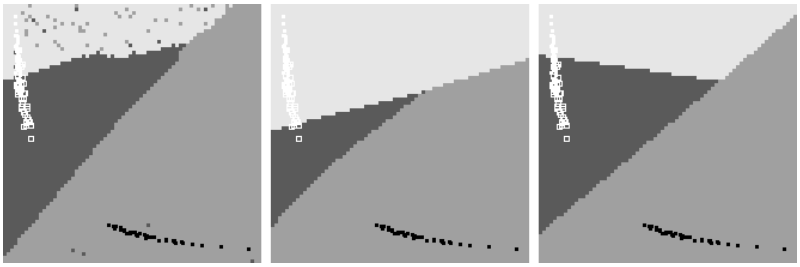


Figure 29.4: Decision regions of the three classifiers on the anisometry × circularity plane (left: KNN; center: MLP; right: SVM. The other two dimensions are fixed at the training-set means; light gray = dot, medium gray = fiber, dark gray = cluster; white squares / black squares / white hollow squares are the training samples of the three classes). KNN mottled, MLP smooth, SVM piecewise curved — a visual portrait of three inductive biases.

Classifier	Train 120 (40/class)	Train 15 (5/class)	Train 6 (2/class)
KNN ($K = 3$, z-score, kd-tree)	100%	98.3%[1]	100%
MLP (4-6-3, 500 iterations)	100%	100%	100%
SVM (NU-SVC, RBF kernel)	100%	100%	100%

The honest conclusion is: **nobody collapses**. Even with only 2 training samples per class, all three classifiers remain almost perfect — because the three classes are far apart in feature space to begin with, and 2 points suffice to stake out each

[1] A single representative run; measured over 5 reruns the accuracy ranged from 91.7% to 100%, and the nondeterminism is not confined to this training-size tier

territory. This is the converse confirmation of the previous section's claim: a task with good separability is insensitive to both classifier choice and sample size; the differences between classifiers only surface in borderline territory and on genuinely hard small-sample problems. Expecting an "easy" acceptance dataset to rank three classifiers against each other is a common methodological mistake in production-line model selection.

One entry in the table must be disclosed honestly: KNN's accuracy is **unstable across runs** — the SDK's kd-tree construction is nondeterministic, so rerunning on the very same data produces occasional flips in KNN's verdicts — measured over 5 reruns the accuracy fluctuated between **91.7% and 100%**, with up to 5 samples flipping in a single run, and the flips are not pinned to any particular training-size tier. The MLP and SVM results, by contrast, are fully reproducible.

29.4 The Ambiguous Zone: Three Semantics of Confidence

Now bring out those 10 fat capsules from the fourth row. Their anisometry of 3.33–5.16 falls in the no-man's-land between dots (1.48) and fibers (7.15); there is no ground truth — what interests us is not "right or wrong" but how three classifiers trained on the same data will rule, and how each of them expresses "how sure it is."

The result is a genuine three-way disagreement: **KNN labels 8 as fiber and 2 as dot; the MLP labels all 10 as dot; the SVM labels 9 as fiber and 1 as cluster.** Same training set, same query points, and three inductive biases deliver three answers: KNN looks locally — the capsules sit closer to the fiber sample cloud, so the neighbor vote tips toward fiber; the MLP's globally smooth surface extrapolates the dots' territory clear across the no-man's-land; the SVM's margin midline assigns most capsules to the fiber side, while the most circular one (anisometry 3.33) gets pushed to the third party — cluster. **What classifiers' disagreement at the boundary exposes is not who is right and who is wrong, but their respective inductive biases** — wherever the training data

offers no coverage, a model can only fill in the blanks according to its own worldview.

Of greater engineering value is how the three express confidence. KNN reports the neighbors' votes and similarities: the 8 samples ruled fiber got unanimous 3-vote verdicts, while one sample ruled dot came out as "dot votes 0.667 (similarity 0.613), fiber votes 0.333 (similarity 0.635)" — the votes and the similarities even contradict each other; the hesitation is written all over its face. The MLP reports soft scores: nearly half the capsules receive a perfect dot=1.000, and most of the rest score above 0.95 (the lowest being 0.703) — a reminder, once again, that soft scores are a byproduct of surface extrapolation: **a high score does not mean high reliability**, and the farther from the training data, the more it must be discounted. The SVM in this SDK is the bluntest of all: it gives only a hard label — no score, no margin distance, not even a channel through which to express "hesitation."

This leads to an iron law of model selection: **a production line that needs rejection or a manual-review safety net must choose a classifier with usable confidence output**. This SDK's SVM outputs hard labels, which means that on such a line it cannot support a "low confidence goes to a human" workflow — no matter how pretty its accuracy.

Rejection is a standard requirement of industrial classification: better to pull out doubtful samples for manual review than to let misclassifications escape. The prerequisite for rejection is that the classifier outputs comparable confidences that can be thresholded — the same mechanism as using minScore to cull low-scoring characters in Chapter 28.

29.5 SciVision Implementation

The three classifiers are provided by `SciSvKNNClassifier`, `SciSvMLPClassifier`, and `SciSvSVMClassifier`, with an identical calling skeleton: initialize → `AddSample` per sample → `Train` → `Classify`.

```
// KNN: no training process; Train merely builds the index
SCIMV::SciSvKNNClassifier knn;
knn.InitializeModel(4, featNames);            // feature dimension + array of feature names
knn.AddSample(featVec, className);            // add samples one by one (4-dim SciVarArray + cl
knn.Train(SCI_KNN_Z_SCORE, SCI_KNN_KDTREE);   // z-score normalization, kd-tree index
knn.Classify(3, SCI_KNN_MAX_COUNT, fv,        // K=3, majority vote
             &cls, &score, &names, &votes, &sims);  // outputs neighbor votes and similarities
```

```
// MLP: 4-6-3 fully connected network
SCIMV::SciSvMLPClassifier mlp;
mlp.LoadModel(path);                          // must be called first! a nonexistent file is fi
mlp.InitializeModel(0, 4, 6, 3);              // mode 0 = feature classification; input/hidden/
mlp.SetNames(featNames, classNames);
mlp.Train(500, 1e-6f, 0.0f, 1, 0);            // max iterations, error tolerance, no regulariza
mlp.Classify(fv, &cls, &score, &names, &scores);    // outputs soft scores for all classes

// SVM: NU-SVC + RBF kernel
SCIMV::SciSvSVMClassifier svm;
svm.LoadModel(path);                          // likewise must be called first
svm.InitializeModel(featNames, 4, SCI_SVM_NU_SVC, 0.01, SCI_SVM_RBF, 1.0, 1.0, 0.0);
svm.Train(1000, 1e-6f, 1);                    // min-max standardization
svm.Classify(fv, &cls, &names);               // hard label only; no scores / margin output
```

The APIs themselves all run, but this group of modules harbors **the most severe batch of SDK defects encountered in this book** — multiple instances of cross-module, process-level state corruption, all pinned down through repeated experiments and recorded here verbatim:

- **KNN `Train` reliably crashes (0xC0000005) after `ManualThreshold`**: once threshold segmentation has been called anywhere in the same process, any subsequent KNN training hits an access violation. This forced this chapter's example to be split into two process runs: `demo.exe extract` handles data synthesis, threshold segmentation, and feature extraction, then writes out `features.csv`; `demo.exe classify` starts a fresh process that reads the CSV and performs training and evaluation. Handing off between feature extraction and classification via a plain text file is currently the only reliable isolation.
- **Training KNN after both the MLP and SVM have been trained yields a silently broken model**: no error, no crash, but the cluster class tests 0/20 — every one wrong. The workaround is to organize the code as "KNN block → MLP block → SVM block," finishing all of one classifier's training and prediction before touching the next.

- **After roughly 100 KNN `Classify` calls, the next KNN `Train` in the same process crashes or produces a broken model** (measured: 10 calls safe, 100 calls crash): within each block, all models must be trained first before batch prediction begins.
- **KNN `Train` with treeType=1 (brute-force search) makes the subsequent `Classify` crash** — only the kd-tree (0) can be used; and the kd-tree in turn is nondeterministic across runs (the root of the ±1 flips in the previous section).
- **`LoadModel` must be called once before the MLP's/SVM's `InitializeModel`**, otherwise error 122706011/122707003 — passing a nonexistent file path is fine; getting back the "new empty model created" code is enough to proceed.
- **Under KNN normalization modes 0/1, the class-name strings returned by `Classify` are occasionally garbled** — with z-score (mode 2) the problem has not been observed.

The methodological significance of this list is no less than its content: a commercial library's stability boundaries are not written in the manual — **they can only be charted through systematic experiment** — hold everything else fixed, introduce modules one at a time, and use probe programs to locate the crash thresholds, the same line of thinking as using golden experiments to reverse-engineer the true behavior of interfaces in Chapter 5. Write the workarounds, together with their full backstory, into the code comments (see the file header of `code/classical_classifiers/main.cpp`), so that those who come after need not step on the same mines.

Industry Case: The Feature Debate in Leather Texture Sorting

A synthetic-leather production line needed to sort finished product into three grades by texture coarseness. The first version of the solution scaled down the entire raw image and fed it straight to an MLP: the "end-to-end" accuracy barely cleared eighty percent, was extremely sensitive to illumination drift, and required retraining every time a lamp was changed. After a post-mortem, the project switched to the classical route: for

each image, first compute four GLCM texture features — energy, contrast, correlation, and homogeneity (see Chapter 25) — then classify with KNN in this four-dimensional space. Accuracy overtook the original solution by more than ten percentage points and remained stable under illumination drift — texture statistics are naturally insensitive to overall brightness changes. Even more popular with the quality inspectors was the interpretability: for every misclassified sample, its 3 nearest-neighbor images could be pulled up for manual comparison, making it obvious at a glance "who" a borderline-grade leather surface resembled. The lesson: on small-data industrial tasks, **good features + a simple classifier** usually beats **raw data + a complex model**.

29.6 Summary

- **Separability comes before the classifier**: the success or failure of a classification problem is largely decided at the feature-selection stage. In this chapter's task, dots and clusters overlap completely in the anisometry × circularity projection and are only separated by area and compactness — when troubleshooting a classification problem, plot the feature scatter first.
- **Three inductive biases**: KNN is local memory (mottled boundaries, verdicts traceable to neighbors), the MLP is a globally smooth function (soft scores, iterative training required), the SVM is maximum margin + kernel trick (the boundary determined solely by the support vectors). On easily separable tasks the three are hard to tell apart — even 2 training samples per class yield near-perfect accuracy; the differences only show in borderline territory.
- **Confidence semantics differ across the board**: neighbor votes + similarities, soft scores, hard labels — 10 borderline capsules made three classifiers, each 100% accurate, hand down three different rulings of 8:2, 10:0, and 9:1. A production line that needs rejection and manual review must choose a classifier whose confidence output is usable and thresholdable.

- **A commercial library's stability boundaries must be charted by experiment**: this chapter's SDK classifier modules suffer cross-module memory corruption, and the workarounds (feature extraction and classification in two separate processes; train everything first, then batch-predict) came from systematic crash-localization experiments, not from the manual.
- For a more systematic theory of classifiers (Bayesian decision theory, feature selection, and novelty detection), see the classification chapters of the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018), together with two standard pattern-recognition textbooks (Duda, Hart, and Stork 2001; Bishop 2006). The founding paper behind each of this chapter's three classifiers is worth returning to in the original: the asymptotic error bound of KNN is given by Cover and Hart (Cover and Hart 1967), the back-propagation that MLP training relies on was established by Rumelhart, Hinton, and Williams (Rumelhart, Hinton, and Williams 1986), and the SVM (with its maximum margin and kernel trick) originates in the support-vector networks of Cortes and Vapnik (Cortes and Vapnik 1995).

Here the recognition part of the book draws to a close. From the deterministic decoding of barcodes, to OCR's closed-set character classification, to this chapter's statistical decisions over an open feature space — three chapters have traversed the spectrum "from table lookup to learning": what encoding rules can govern goes to the decoder, what rules cannot govern goes to features and classifiers, and the borderline territory that classifiers cannot govern is left to confidence measures and manual review.

Part VIII

3D Imaging

This part covers the main imaging principles for acquiring 3D data: stereo vision, structured light, laser triangulation, photometric stereo, phase measuring deflectometry (PMD), and confocal imaging and focus variation, along with the scenarios each technique suits best.

30 Overview of 3D Imaging

Everything in the preceding twenty-nine chapters rests on a single premise: a two-dimensional image. A grayscale image can tell you "where it is dark and where it is bright", and armed with the first seven parts of this book you can already squeeze that information to its limit — localizing to 0.02 pixel, measuring a width of 5 micrometers, catching a defect covering 0.1% of an area. But there is a class of problems that a 2D image is fundamentally powerless against: camera imaging is a single **projection**, and at the instant the three-dimensional world is flattened into a plane along the optical axis, the height information is lost forever. A cold-soldered solder ball and a full one may look like two nearly identical bright blobs from directly above; whether the seam between two pieces of metal on a phone's middle frame is flush (the step height) is, in a top-down view, one and the same line.

Industry's wish list for 3D is very concrete: the **coplanarity** of connector pins — whether the tips of dozens of pins lie in the same plane, where a 0.05 mm difference means they will not seat into the pads; the **step height** and **gap** of an assembly; the **volume** of dispensed glue and solder paste — how much glue there is directly determines bond strength, and volume is area times height, of which 2D can give only the first half; the **depth** of scratches and dents — the very same scratch, 2 m deep, can ship, while at 20 m it must be scrapped, yet in a grayscale image the two may look equally dark. What these judgments share is that **the quantity being measured is height itself**; any two-dimensional method can only guess at it indirectly, and the cost of guessing wrong is paid by the production line.

And so this book enters the third dimension. Part VIII spans seven chapters: this one is the map, and each of the following

six covers a single 3D imaging technique — stereo vision, structured light, laser triangulation, photometric stereo, deflectometry, and focus methods. None of the six is inherently better or worse than the others; there is only the matter of fit: for one and the same height-measurement need, change the surface material and the optimal solution may be entirely different. The task of this chapter is to make "how to choose" clear: first to unify the language of 3D data, then to give each technique a physical intuition, and finally to land on a single selection table and a few real decision paths.

A note up front: this is a theory chapter (an overview of technique selection), with no standalone companion code project and no figures; the principle experiments, real data, and implementation code for each technique are given in its own dedicated chapter. The core deliverable of this chapter is the selection comparison table in Section 30.3.

30.1 The Forms of 3D Data

Before discussing techniques, let us settle what "3D data looks like". Industrial 3D vision has three principal representations, and they carry different amounts of information and suit different algorithms.

A **depth map**, also called a **range image**, is the representation closest to 2D thinking: it is still an "image" on a regular grid, except that each pixel stores not a gray value but the height or distance at that position (typically a 32-bit float in mm). Nearly all the tools of the earlier chapters — filtering, thresholding, blobs, calipers — can be carried over almost untouched, except that "gray-value difference" becomes "height difference". This representation is also called **2.5D**: each (x, y) position holds only **one** height value, expressing "the surface seen from a particular viewpoint" rather than a complete 3D solid — the inner wall of a cup, or the side and bottom faces of a part, simply do not exist in a single-view depth map.

A **point cloud** is an unordered set of three-dimensional coordinate points $\{(x_i, y_i, z_i)\}$, optionally carrying attributes such as normals and color. It does not rely on a regular grid and

can express a surface of arbitrary viewpoint and arbitrary topology; merging multi-view data can approximate a true 3D model. The price is the loss of the implicit structure "the neighbor lives in the adjacent pixel" — finding nearest neighbors now relies on a spatial index (a kd-tree), and algorithm costs rise overall. A **mesh** restores connectivity on top of the point cloud — describing surface topology with triangular facets, it is the common language of CAD comparison and visualization.

The three convert into one another: back-projecting a depth map pixel by pixel using the calibration parameters yields an ordered point cloud; surface reconstruction from a point cloud yields a mesh; resampling a mesh along some direction returns to a depth map. The mainstream path of industrial metrology is "the sensor outputs a depth map or an ordered point cloud → convert to an unordered point cloud when necessary for registration and comparison". The outputs of the six technique chapters that follow are basically depth maps; the data structures and operations of point clouds are developed systematically starting from Chapter 37.

30.2 The Physical Principles of the Six Techniques

"2.5D or true 3D" is the first clarifying question when selecting a solution. Most inline inspection (coplanarity, step height, volume) is fine with single-view 2.5D; only when a complete contour is needed (such as reverse engineering or comparison of complex curved surfaces) is multi-view stitching required, and its cost and cycle time must be budgeted separately. Mistaking the need for true 3D can multiply the budget several times over.

The six techniques look wildly different, but the underlying logic is a single sentence: **height itself cannot be imaged directly, but using a known geometric or optical constraint it can be transformed into some measurable image quantity** — disparity, phase, displacement, shading, slope, or sharpness — and then inverted back. Let us go through, one by one, what that "image quantity" is.

Stereo vision (Chapter 31): the image quantity is **disparity**. Two cameras separated by a **baseline** shoot simultaneously, and for the same spatial point the difference in column coordinates between the left and right images is inversely proportional to depth — near points "jump" a lot, far points "jump" little, the same mechanism by which human eyes judge distance. It is a **passive** technique that projects no light and

has the simplest structure; but "the same point corresponds across the two images" must be established by matching the image content itself, so it **depends on surface texture** — a smooth, uniform surface (a white wall, polished metal) yields no correspondences, and the depth map is left with stretches of holes.

Structured light (Chapter 32): the image quantity is **phase**. If passive matching fears the lack of texture, then actively "print" texture onto the surface — a projector casts sinusoidal fringes onto the object, and a camera observes from another angle: where the surface rises and falls, the fringes are distorted, and the amount of distortion is encoded in the phase of the fringes. Phase-shifting can solve for the phase independently at each pixel, giving high spatial resolution and single-shot area imaging; it is the workhorse for full-field measurement of diffuse surfaces.

Laser triangulation (Chapter 33): the image quantity is **the pixel displacement of the laser line**. A line of laser light strikes the surface at a known angle, and a camera views that bright line from another angle: the higher the surface, the larger the offset of the bright line in the image — two angles and one side of the triangle are known, so the height follows. A single exposure yields only one height profile, and the full surface is stitched line by line from the **scanning** motion of the workpiece or sensor; it is therefore naturally suited to conveyor and robot-arm scenarios, and is the most widely installed solution for industrial inline 3D inspection.

Photometric stereo (Chapter 34): the image quantity is **shading**. Camera and workpiece both stay still while light is cast in turn from several known directions for several exposures: faces oriented toward the light are bright, faces turned away are dark, and combining the several shadings solves for the **surface normal** at each pixel. Note that it measures the normal (slope) directly rather than height; absolute height must be recovered by integration, with errors that accumulate. But it is extremely sensitive to **microscopic relief** — scratches, dents, embossed characters, and the like, features with "tiny height change but drastic slope change", show up in vivid detail in the normal map.

Phase measuring deflectometry (PMD, Chapter 35): the image quantity is **the phase distortion of the reflected fringes**, corresponding to surface **slope**. The previous techniques all assume a diffuse surface and fail en masse on specular surfaces (phone glass, polished metal, car paint) — the light cast onto them bounces off by the law of reflection, and the camera sees a mirror image of the environment rather than the surface itself. PMD turns this around and uses it: it lets the fringes displayed on a screen enter the camera **after specular reflection off the surface**, and the slightest change in surface slope distorts the fringes in the reflected image noticeably — the more "perfect" the mirror, the more sensitive the measurement. It is the dedicated solution for specular and near-specular surfaces.

Focus variation and confocal (Chapter 36): the image quantity is **sharpness**. A lens has finite depth of field, and the object surface is in focus only where it lands on the focal plane. Scanning layer by layer along the optical axis to acquire an image stack, for each pixel one finds the layer at which it appears "sharpest", and the Z position of that layer is the height at that point. Confocal uses a pinhole to physically reject out-of-focus light, achieving sub-micron axial resolution. This family is slow (the whole height range must be scanned) and has a small field of view, but its accuracy is the best of the six, targeting roughness and microstructure measurement at the microscopic scale.

30.3 Selection Decisions

Selection has four core dimensions. **Surface type**: diffuse, specular, or transparent? This is the hardest constraint and eliminates half the candidates outright. **Measurement range and resolution**: is the field of view at the millimeter scale or the meter scale? Does the height resolution need to be micron or sub-micron? The two are roughly proportional — the larger the field of view, the coarser the single-point accuracy. **Speed and motion pattern**: single-frame imaging, workpiece scanning required, or multiple frames in place? This

Classifying by "image quantity" also lets you anticipate each technique's failure mode: those relying on disparity fear the lack of texture, those on phase fear glare and interreflection, those on displacement fear occlusion, those on shading fear non-Lambertian surfaces, those on slope recognize only mirrors, and those on sharpness fear transparency and slow speed. The first round of elimination in selection is often simply matching failure modes against the workpiece surface.

decides whether the system can keep up with the cycle time and whether it can measure a workpiece in motion. **Ambient-light immunity**: active projection guarantees a signal-to-noise ratio, while passive techniques must be especially careful amid stray light on the shop floor.

The six techniques compare as follows (the numbers are order-of-magnitude figures for typical industrial products; specific models vary):

Table 30.1 Selection comparison of the six 3D imaging techniques

Dimension	Stereo vision	Structured light	Laser triangulation	Photometric stereo	PMD	Focus / co
Image quantity	Disparity	Phase	Line displacement	Shading → normal	Reflected phase → slope	Sharpness
Active / passive	Passive	Active	Active	Active	Active	Active
Diffuse surface	Needs texture	Excellent	Excellent	Excellent	Not possible	Excellent
Specular surface	Not possible	Poor	Poor	Poor	**Specialty**	OK
Transparent surface	Not possible	Poor	Poor	Not possible	OK (surface)	Excellent cal)
Typical FOV	0.1–10 m	0.05–1 m	0.01–1 m (wide) × scan length	Same as 2D FOV	0.05–0.5 m	0.1–5 mm
Height resolution	mm scale	5–50 µm	1–20 µm	Normal sensitivity / height diff.	Slope, µrad scale	0.01–1 µm
Imaging pattern	Single frame	Multi-frame (fringe casts)	Line-by-line scan	Multi-frame (switch light)	Multi-frame (screen fringes)	Scan along
Moving workpiece	Yes	Hard	**Naturally suited**	Hard	Hard	No
Ambient-light immunity	Weak	Medium	Strong (narrowband filter)	Medium	Medium	Strong
Typical applications	Logistics volume, robot guidance	Solder paste / glue, full inspection	Step height, burrs, weld seams	Scratches, embossed chars	Glass / paint surface shape	Roughness crostructu

Let us walk a few typical scenarios through the table. **Phone middle-frame step height** (highly reflective metal, on the order of 0.01 mm, inline cycle time): the surface rules out stereo and structured light, leaving PMD and laser triangulation as candidates — the former measures slope and

is keener on step-like jumps such as step height, while the latter, combined with shading and angle optimization, can often be made to work too, and the two frequently compete head-to-head in trials. **PCB solder-paste inspection** (diffuse paste, requiring area + height + volume, tight cycle time): structured light covers the whole board in a single area shot and is the mainstream choice for SPI equipment; scanning laser triangulation competes on equal footing in high-precision cases. **Battery electrode-foil burrs** (continuous web feed, burr height tens of microns): the workpiece itself is in motion, so line-by-line scanning laser triangulation is the natural answer. **Polished-metal scratches** (specular, micron-deep, judging depth rather than presence): PMD reads the slope jump and photometric stereo reads the normal jump, one of the two or a combination of them, while the other techniques have essentially no foothold.

As you can see, selection is not "pick whichever has the highest accuracy" but rather: surface type makes the first cut, accuracy and field of view make the second, speed and motion pattern make the third, and the remaining candidates go to trial.

30.4 Common Engineering Elements

The six techniques each have their own chapter, but a few matters are common to all 3D systems, and we set up the framework here first.

Calibration. The raw quantity each technique measures is an image quantity — disparity, phase, pixel displacement — and converting it into a physical height in millimeters relies on calibration in every case: stereo must calibrate the intrinsics and extrinsics of the two cameras, laser triangulation must calibrate the pose of the light plane relative to the camera, structured light must calibrate the projector, and PMD must calibrate the screen. The camera model and calibration methodology established in Chapter 5 are the foundation of all of Part VIII; one might say that **the**

accuracy ceiling of a 3D system is written down already at calibration time.

Occlusion and missing data. Any technique whose projection direction differs from its observation direction (the entire triangulation family) has dead zones where "it can be projected onto but not seen, or seen but not projected onto", manifesting as holes in the depth map; steep walls, deep holes, and reflective flyspecks also produce invalid data. Whether and how to fill holes (conservatively flag as invalid, or interpolate from the neighborhood) is an engineering decision that affects the credibility of later measurements, and the system's handling is left to Chapter 38. The multi-view stitching needed when a single view is not enough — unifying point clouds from different viewpoints into one coordinate system — is borne by the registration algorithm of Chapter 39.

The language of accuracy. Chapter 20 distinguished **repeatability** from **accuracy**: the former is the spread of repeated measurements of the same workpiece, the latter the degree to which the measurement mean deviates from the true value. This methodology carries over to 3D unchanged — to evaluate a 3D sensor, repeatedly measure a standard gauge block or step gauge: repeatability corresponds to height noise (often expressed as a Z-direction), and accuracy is verified against a traceable standard artifact. A vendor's nominal "resolution" is often the minimum resolvable quantity under ideal conditions, which is a different matter from the repeatability you can achieve with your workpiece at your site — and this is exactly the reason for the "trial verification" in the case below.

Industry Case: A Step-Height Measurement Selection Journey

A structural-parts plant required a 0.02 mm step-height judgment on the assembled metal middle frame, and the integrator's selection went three rounds. The first round chose stereo vision on cost: it was the cheapest, but the middle-frame surface was uniform and textureless, stereo matching failed over large areas, and the depth map had holes exactly at the critical seam — eliminated outright. The second round switched to structured light: the diffuse regions gave beautiful data, but near the seam was precisely a highly

reflective chamfer, the phase was stained into patches of noise at the glare, and the 0.02 mm judgment line drowned in 0.05 mm of noise. The third round tried laser triangulation: a short-wavelength narrowband laser paired with a filter was chosen, the incidence angle was tuned to avoid the specular reflection's main lobe, and a shroud was added to isolate the shop's overhead lights; a step gauge measured a Z-direction repeatability of 3 m in practice, ample margin, and the solution went live. The project manager's takeaway at the postmortem: selection is not scoring against a spec sheet but eliminating one by one along "surface → accuracy → cycle time"; and, **trial verification always precedes purchase** — the spec sheets of the first two rounds both "met the requirements".

30.5 Summary

- A 2D image is a projection of the three-dimensional world, and height information is lost in principle; tasks where "the quantity being measured is height" — coplanarity, step height, volume, scratch depth — must use 3D imaging.
- 3D data has three forms — depth map (range image, 2.5D, one height per pixel), point cloud, and mesh — convertible into one another; most inline inspection is fine with single-view 2.5D, while true 3D requires multi-view stitching, costed separately.
- The six techniques share one logic: **using a known geometric or optical constraint to turn height into a measurable image quantity** — disparity (stereo), phase (structured light), line displacement (laser triangulation), shading → normal (photometric stereo), reflected phase → slope (PMD), sharpness (focus methods); the image quantity also determines each one's failure mode.
- Selection eliminates in three cuts along "surface type → accuracy and field of view → cycle time and motion pattern" (see the selection table in this section): full-field diffuse goes to structured light, inline scanning to laser triangulation, specular to PMD, micro-relief to photomet-

ric stereo, microscopic accuracy to confocal, and large-scene low-cost to stereo; candidate solutions must be trial-verified before purchase.

- All 3D systems share three things in common: calibration sets the accuracy ceiling (Chapter 5), occlusion and missing data need an explicit handling strategy (Chapter 38, Chapter 39), and accuracy assessment follows the language of repeatability and accuracy (Chapter 20).

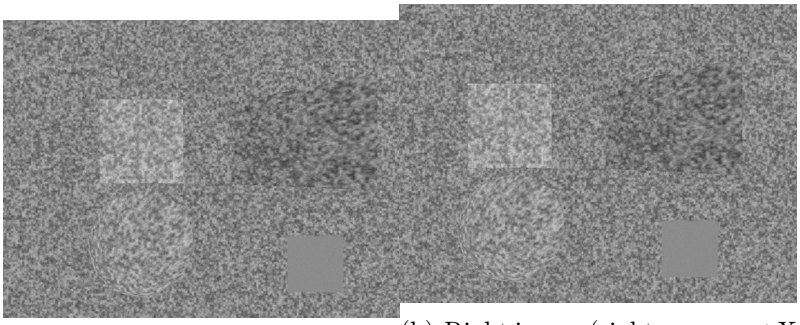
The next six chapters unfold one by one in the order of the table: Chapter 31 covers epipolar geometry and stereo matching, Chapter 32 covers phase shifting and unwrapping, Chapter 33 covers light-plane calibration and profile stitching, Chapter 34 covers normal solving and integration, Chapter 35 covers specular fringe reflection, and Chapter 36 covers sharpness evaluation and the focus stack. For a systematic treatment of 3D imaging, see further the chapters on 3D imaging and reconstruction in the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018); Szeliski's computer vision textbook (Szeliski 2022) offers a unified view of passive and active 3D methods from the perspective of multi-view geometry and depth estimation.

31 Stereo Vision

Humans have two eyes, and so we can tell at a glance which object is closer — the brain converts the tiny positional difference of the same target on the two retinas into distance. Machine vision can borrow this mechanism wholesale: shoot with two cameras separated by a certain distance at the same time, and the same spatial point lands at different horizontal positions in the two images; this positional difference is the **disparity**, and disparity directly encodes depth. Binocular stereo vision is the technique of inferring the depth of every pixel from a pair of images. Among the several 3D imaging routes listed in Chapter 30, it is the one closest to the human eye and also the most "passive": it projects no energy onto the scene and relies only on ambient light, hence low power, no mutual interference, and the ability to work at long range. The price is an inescapable prerequisite — **the measured surface must carry texture of its own**, because finding disparity is essentially finding the correspondence of the same patch of pattern across the two images, and without texture there is nothing to correspond. This chapter starts from epipolar geometry, proceeds through block matching, reliability checking, subpixel refinement and windowing, and finally arrives at depth reconstruction, throughout using one synthetic rectified stereo pair (Figure 31.1).

31.1 Epipolar Geometry and Disparity

When two cameras look at the same point, the geometry is not entirely unconstrained. The spatial point and the two camera optical centers jointly determine a plane, which intersects the two image planes in a pair of **epipolar lines** — this is the core conclusion of **epipolar geometry**: the corresponding point in the right image of a given pixel in the left image must



(a) Left image (left camera, origin) (b) Right image (right camera, +X 60 mm)

Figure 31.1: A synthetic rectified stereo pair. The scene contains a background plane, a raised box, a slanted ramp on the right, a spherical cap at the lower left, and a deliberately designed textureless gray patch at the lower right corner. The whole content of the right image is shifted left relative to the left image, and the amount of shift is the disparity — near objects (box, spherical cap) move more, the far background moves less.

lie on its corresponding epipolar line, rather than roaming all over the image. This reduces a two-dimensional search to a one-dimensional one. Engineering goes one step further: through **rectification** a projective transform is applied to the two images so that all epipolar lines are strictly parallel to the image rows and their row indices align one-to-one. After rectification, the corresponding point of a pixel on row y of the left image must lie on the same row y of the right image, offset only in the x direction. This is exactly the state of this chapter's stereo pair: finding correspondence degenerates into a one-dimensional slide along the same row.

Let the two camera optical axes be parallel, separated by the **baseline** B, with focal length f (in pixels); for a point at depth Z, the difference in horizontal coordinate between the left and right images is the disparity d. From similar triangles we get $d/f = B/Z$, that is

This example's stereo pair is already in the rectified state at the synthesis stage (the two cameras differ only by a translation along X and share identical intrinsics), so in-row search can be done directly. A real system must first perform the binocular calibration of Chapter 5, solving for the relative pose and distortion of the two cameras, and then compute the rectification mapping accordingly — rectification quality directly decides the success or failure of subsequent matching: even half a pixel of residual vertical misalignment makes block matching systematically worse.

$$Z = \frac{fB}{d}. \tag{31.1}$$

Disparity is inversely proportional to depth: the nearer the object the larger the disparity, the farther the smaller. In this example $f = 600$ px and $B = 60$ mm, so $fB = 36000$ mm $\cdot$ px, and the ground-truth disparity falls within 36.0–51.4 px (corresponding to Z from 1000 mm to 700 mm). In Figure 31.1 the near objects shift left noticeably while the background barely moves, which is exactly the intuitive manifestation of this inverse relationship.

The inverse relationship also has a far-reaching corollary. Differentiating Equation 31.1 gives the depth error corresponding to one pixel of disparity error, namely the **depth resolution**

$$\Delta Z = \frac{Z^2}{fB}. \tag{31.2}$$

It grows with Z^2 — this is the mathematical essence of "stereo is less accurate farther away." In this example at $Z = 1000$ mm, each 1 px of disparity corresponds to **27.8 mm** of depth; while at $Z = 800$ mm it is only 17.8 mm, and at $Z = 700$ mm it drops further to 13.6 mm. The same disparity measurement error gets amplified into several times the depth error at a distance. This formula also points to two directions for improving accuracy: increase fB (longer focal length or longer baseline), or measure disparity to subpixel accuracy — the latter being the subject of Section 31.4.

Figure 31.2 is the ground-truth disparity field of this scene (obtained per pixel from the known depth via Equation 31.1), and all subsequent matching results will be measured against it. Note that the box and spherical cap (near objects) are brighter overall (large disparity), the background is darker (small disparity), and the ramp shows a continuous gradient from near to far — this is exactly the pictorial presentation of the inverse relationship in Equation 31.1, and also the target that block matching ought to approach.

The baseline B is a typical trade-off quantity. By Equation 31.2, the longer the baseline the higher the depth resolution; but the longer the baseline, the larger the viewpoint difference between the two cameras, and the larger the **occlusion** regions blocked by the foreground and visible to only one eye, while the same surface suffers more severe perspective deformation across the two images and is harder to match.

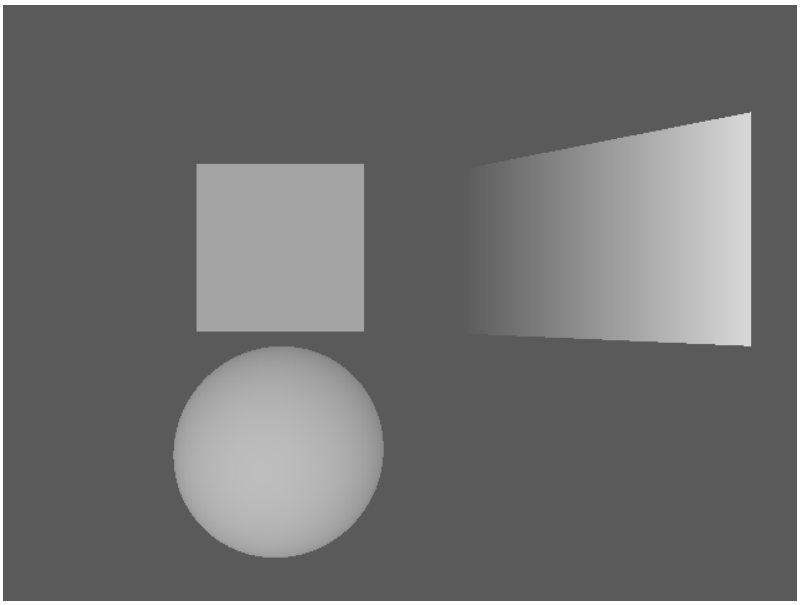


Figure 31.2: Ground-truth disparity field back-computed from the known depth (bright = large disparity = near). It is the yardstick for evaluating all subsequent matching results: the box/spherical cap are bright, the background is dark, and the ramp is a continuous gradient.

31.2 Block Matching

After rectification, finding disparity is simply: for each pixel of the left image, slide along the same row of the right image and find the most similar position. "Similar" needs a cost function. The most naive is the **sum of absolute differences (SAD)** — take a small window centered on the pixel, compute the absolute difference of left-right gray values point by point and sum them; the smaller the cost the more alike. This is of the same origin as the template matching of Chapter 16, only the template is replaced by "a local window of the left image" and the search range is constrained by epipolar geometry to within one row. More robust to illumination changes is **normalized cross-correlation (NCC)**, but it is heavier to compute. This example uses SAD and restricts the search to the disparity interval $[30, 56]$ px (derived from the scene's known depth range plus margin).

Computing the window cost for every pixel and every candidate disparity yields a three-dimensional **cost volume** $C(x, y, d)$. Taking the position of minimum cost along the d dimension gives the integer disparity of that pixel — this step is called "winner-take-all." Figure 31.3 is the raw integer disparity result of an 11×11 window: the shapes of the box, ramp, and spherical cap all emerge correctly, near objects bright (large disparity), background dark (small disparity). But look closely at the ramp and the spherical cap and you find obvious **quantization banding** — depth that ought to transition smoothly is cut into a staircase, exactly the discretization trace of integer disparity: the true disparity is a continuous fraction, while winner-take-all can only output integers.

Quantitatively, this 11×11 SAD reaches 91.8% valid rate over the whole image, mean absolute error (MAE) of 0.362 px, and a bad-pixel rate (error > 1 px) of 3.93%. If we count only non-occluded regions, the MAE drops to 0.299 px — confirming that one major source of error is occlusion: background pixels blocked by the foreground simply have no correspondence in the right image, and winner-take-all can only assign them a wrong minimum-cost match. In this

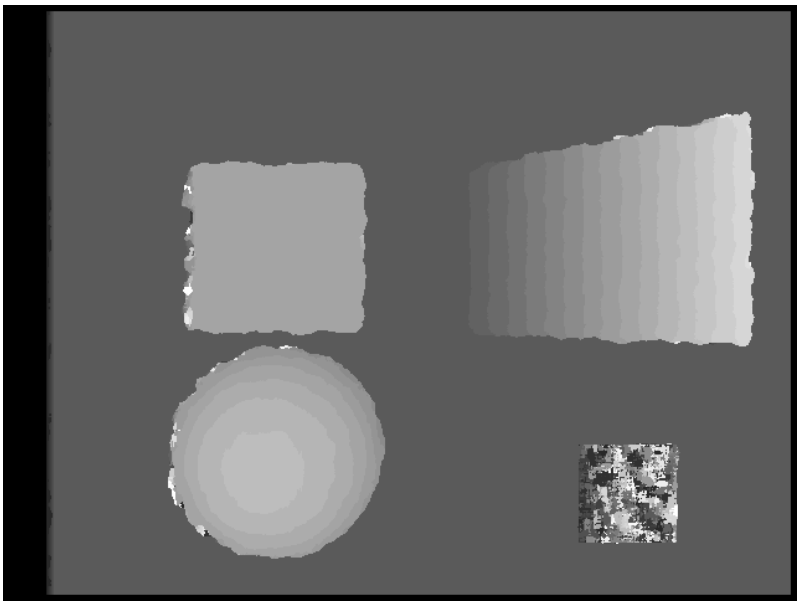


Figure 31.3: Raw integer disparity of 11×11 SAD block matching (near objects bright, far dark). Staircase quantization banding is visible on the ramp and spherical cap, a direct consequence of integer disparity discretization; the textureless patch at the lower right shows chaotic random disparity.

example the ground-truth occlusion accounts for 0.92%, concentrated in the background band along the left edge of near objects. How to identify these unreliable disparities is the task of the next section.

31.3 Failure and Reliability

Winner-take-all will always give a "best" disparity, even where the region has no reliable correspondence at all. A naive yet effective discrimination means is the **left-right consistency check**: compute a disparity map d_L with the left image as reference, then compute another d_R with the right image as reference, and cross-check — if the disparity of left-image pixel x is $d_L(x)$, its corresponding point in the right image is $x - d_L$, then $d_R(x - d_L)$ should be almost equal to $d_L(x)$. Pixels where the two disagree (threshold taken as 1 px in this example) are judged unreliable and discarded. Occlusion regions are naturally caught by this check: the "corresponding point" of an occluded pixel actually belongs to the foreground, the two matches each tell their own story, and they are bound to disagree.

Figure 31.4 is the disparity map after checking: most of the occlusion band (left edge of near objects) and the textureless patch have been culled to black (invalid), and the retained disparities are cleaner. The price is that the whole-image valid rate drops from 91.8% to 88.6%, but the quality of the retained pixels improves substantially — MAE improves from 0.362 px to 0.134 px, and the bad-pixel rate is squeezed from 3.93% down to 1.26%. The reliability check does not create accuracy; it merely honestly marks out the places where "I don't know," which in industry is often far more useful than a seemingly dense but actually adulterated disparity map.

The most illustrative case is the 90×90 px **textureless patch** at the lower right corner — it is a deliberately designed failure region. In Figure 31.3 it shows a mess of random disparity, and the raw statistics reveal: this region has a raw valid rate as high as 100%, yet 82.2% of it is bad pixels. The reason is simple: every position on a uniform gray patch looks identical,

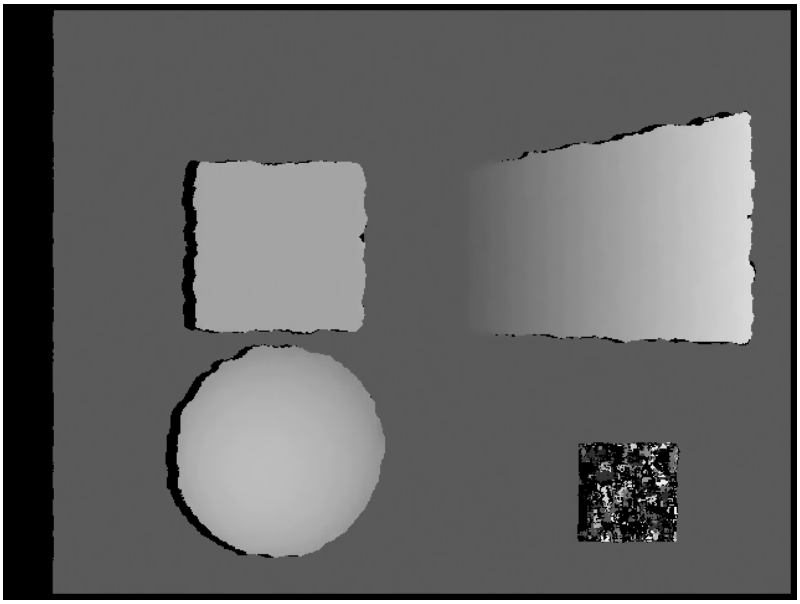


Figure 31.4: Disparity map after the left-right consistency check. The occlusion band and textureless patch are culled to black (invalid); the MAE of the retained pixels drops from 0.362 px to 0.134 px. Honestly marking out the regions that "cannot be reliably measured" is a key step toward making stereo vision engineering-usable.

the SAD cost curve is nearly flat across the entire search range, and the "minimum" picked by winner-take-all is determined purely by sensor noise — a thoroughly random solution. Even more telling is what happens after the left-right check: only 43.3% of the pixels of the textureless patch survive, and among the survivors still 82.5% are bad pixels. In other words, even when two random matches "happen" to agree, what they agree on is still wrong — **no texture, no stereo**.

The lesson of the textureless patch directly explains the motivation of actively projecting texture in Chapter 32: since passive stereo is helpless on textureless surfaces, then actively project a sheet of artificial random speckle or coded fringes onto the scene, manufacturing texture available for matching. This is also the core idea of "active stereo" sensors — the geometric framework of stereo is unchanged, only a projector supplies the prerequisite of texture.

31.4 Subpixel and Windowing

The quantization banding of integer disparity (Figure 31.3) comes from winner-take-all being able to take only integers. The remedy is just like the subpixel edge localization in Chapter 14 and Chapter 20: at the integer disparity d_0 of minimum cost, take the three cost values C_{-1}, C_0, C_{+1} of it and its left and right neighbors $d_0 \pm 1$ and do **parabolic interpolation**; the offset of the parabola vertex relative to d_0 is

$$\delta = \frac{C_{-1} - C_{+1}}{2\,(C_{-1} - 2C_0 + C_{+1})}, \tag{31.3}$$

and the subpixel disparity is $d_0 + \delta$. Three samples determine one parabola, the vertex formula is closed-form, and the computational cost is negligible — once again, accuracy almost for free. The effect is most intuitive in the ramp region (where the true value is a continuous fractional disparity): there the integer-disparity MAE is 0.252 px, and after subpixel interpolation it plunges to **0.070 px**. The whole-image non-occluded error histogram also moves accordingly: the fraction of pixels with $|\text{err}| \le 0.125$ px rises from 80.4% for integer to 91.6% for subpixel. The whole-image MAE improves from 0.362 to 0.342 px, and the non-occluded one from 0.299 to 0.278 — the quantization banding is smoothed away.

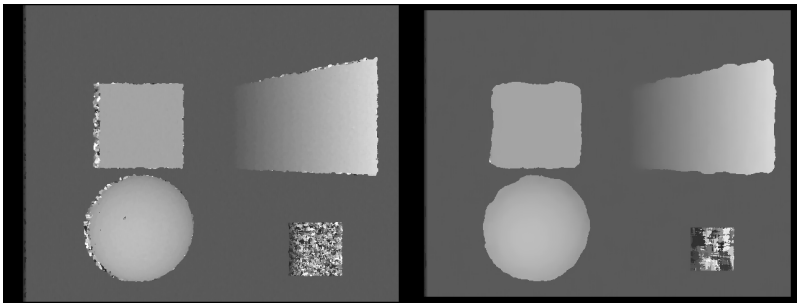
Another knob that must be tuned to the scene is the **window size**. A large window admits more texture and averages noise more fully, but it presumes that "the entire window belongs to one depth," and at depth discontinuities (object edges) it smears foreground and background together. The table below contrasts three window sizes — 5×5, 11×11, 21×21 — in the non-occluded region (subpixel disparity):

Table 31.1: Window size trade-off: a large window is more accurate overall but worsens at edges (non-occluded, subpixel disparity)

Window	MAE (px)	Bad-pixel rate	Edge-ring bad pixels	Textureless bad pixels
5×5	0.336	3.69%	3.95%	90.5%
11×11	0.278	3.38%	4.26%	86.7%
21×21	0.251	3.24%	5.17%	65.5%

Read this table region by region. The overall MAE decreases as the window grows (0.336 → 0.251), because a large window suppresses noise more aggressively; but the bad-pixel rate of the **edge ring** (12 px on each side of the box boundary) worsens in the opposite direction (3.95% → 5.17%) — this is **edge bleeding**: a large window also counts background pixels outside the foreground edge, so the edge is "fattened" by the foreground. Figure 31.5 contrasts the two extremes 5×5 and 21×21: win5 is noisier with sharp edges, win21 is smooth and clean but the contours of the box and ramp are clearly swollen. Interestingly, the textureless bad-pixel rate drops from 90.5% for win5 to 65.5% — the large window borrows the externally adjacent texture in too, scraping together a bit of matchable content for the otherwise hopeless gray patch, but this is borrowed and not trustworthy.

Edge bleeding can be quantified to the specific pixel. Measuring the disparity-crossing position of the box edge on row $y = 195$: the box left edge has true value $x = 155$, and the three windows measure 148/151/152 — the larger the window the more it biases to the outer left (foreground



(a) 5×5 window (noisier, sharp edges)

(b) 21×21 window (smooth, but foreground fattened)

Figure 31.5: The dilemma of window size. A small window preserves edges but is poor at noise rejection; a large window is smooth but bleeds at the edges, with visible swelling of the box and ramp contours. There is no universal window size; it must be compromised per scene between "measurement accuracy vs. edge fidelity."

expansion); the right edge has true value $x = 290$, measured 291/292/293 — likewise swelling outward. Each step up in window size expands the foreground edge by roughly half a window width. This explains why industry often places extra distrust on the disparity near foreground edges, or simply switches to algorithms with adaptive windows or global smoothness constraints.

31.5 Depth Reconstruction

With a reliable disparity map, applying Equation 31.1 per pixel as $Z = fB/d$ gives the depth map (Figure 31.6). This map is essentially a **2.5D range image** — it records the distance to the nearest surface in each pixel direction as seen from the camera, rather than a complete three-dimensional model (no data for the back side or occluded places), consistent with the definition of 2.5D in Chapter 30. To go further and obtain metric 3D points, one still needs the camera intrinsics to back-project each (x, y, Z) into (X, Y, Z), which enters the point-cloud realm of Chapter 37.

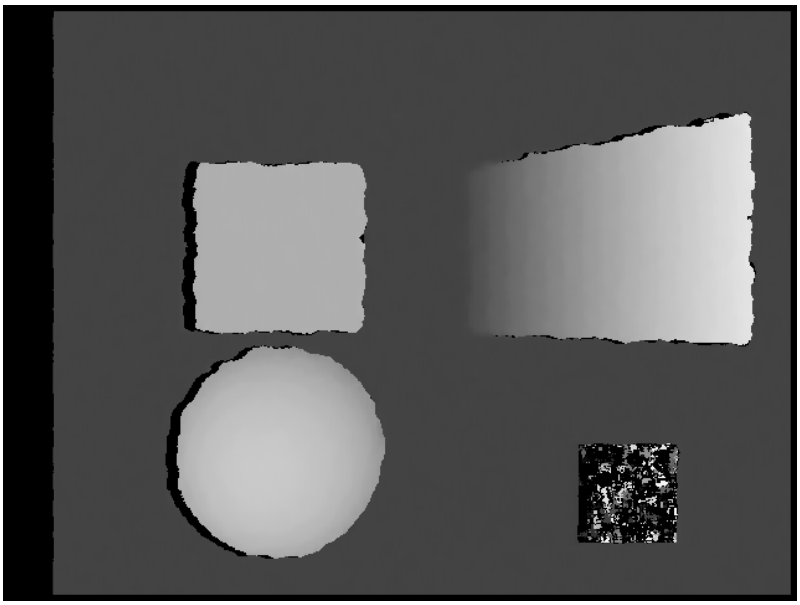


Figure 31.6: Depth map converted from the disparity map via $Z = fB/d$ (near bright, far dark). This is a 2.5D range image: it records only the distance to the nearest surface in each line-of-sight direction. The reconstructed box height is 200.1 mm (true value 200).

In terms of accuracy, the measurement results are convincing: the background median depth is 1000.0 mm (true 1000), the box face is 800.0 mm (true 800), so the reconstructed box height is **200.1 mm, against a true value of 200 mm** — an error of a few tenths of a millimeter. Here the subpixel dividend of Section 31.4 is exactly recovered: by Equation 31.2, the depth resolution at $Z = 1000$ mm is 27.8 mm/px, and with only integer disparity the depth would be nailed to a ~28 mm staircase, with no way to measure the 200 mm box height to one decimal place. But the 0.1 px subpixel accuracy raises the effective depth resolution to **2.78 mm** (×10), and only then can the box height be pinned to 200.1 mm. Subpixel is not icing on the cake; it is the watershed of whether stereo can do metric measurement.

31.6 SciVision Implementation

It must be stated honestly: the SciVision SDK **provides no binocular stereo matching library** — there is no ready-made API for rectification, cost volume, disparity solving, or left-right checking. Therefore the companion project of this chapter `code/stereo_vision/` is a **self-contained C++ implementation**, with the SDK used only for image I/O (`SciImage::SaveImage` outputs PNG). Below are three core code segments.

Winner-take-all over the cost volume (SAD accelerated by box filtering with integral images; the integral-image details are omitted here, illustrating only the optimal-disparity solve and subpixel):

```
int best = BIG, bi = -1;                    // find the minimum cost along the disparity dimensi
for (int di = 0; di < ND; ++di) {
    int c = cost[((size_t)di * H + y) * W + x];
    if (c < best) { best = c; bi = di; }
}
m.id[i] = D0 + bi;  m.d[i] = (float)(D0 + bi);  m.valid[i] = 1;
if (bi > 0 && bi < ND - 1) {                // parabolic subpixel interpolation
    double cm = cost[((size_t)(bi-1)*H + y)*W + x];
    double cp = cost[((size_t)(bi+1)*H + y)*W + x];
```

```
    double denom = cm - 2.0 * best + cp;
    if (denom > 1e-9) {
        double offd = 0.5 * (cm - cp) / denom;   // i.e. eq. @eq-sv-subpix
        if (offd > 0.5) offd = 0.5; else if (offd < -0.5) offd = -0.5;
        m.d[i] = (float)(D0 + bi + offd);
    }
}
```

Left-right consistency check (cross-checking the left and right match results):

```
auto bmR = blockMatch(right, left, false, 11);   // match once more with the right image as re
for (int y = 0; y < H; ++y)
    for (int x = 0; x < W; ++x) {
        int i = y * W + x;
        if (!bm11.valid[i]) continue;
        int xr = x - (int)std::lround(bm11.d[i]);     // corresponding point of the left-image
        if (xr < 0 || xr >= W || !bmR.valid[y*W+xr]) continue;
        if (std::fabs((double)bm11.d[i] - bmR.d[y*W+xr]) <= 1.0)  // keep only if consistent
            lrValid[i] = 1;
    }
```

`blockMatch(left, right, true, 11)` and `blockMatch(right, left, false, 11)` solve d_L and d_R respectively, `win=11` is the window edge length, and `D0=30`, `D1=56` frame the search interval. This pattern of "the algorithm living outside the SDK" is no accident: industrial stereo is extremely sensitive to compute and latency, and **real products mostly push stereo matching down into a dedicated sensor or FPGA** — disparity solving is a regular, massively parallelizable per-pixel operation, naturally suited to a hardware pipeline, and belongs to a different layer than the "call-the-library" paradigm of a general CV SDK. The place for a library like SciVision is **after** the disparity/depth map is produced: use its filtering, morphology, blob, and measurement tools to process the depth map and complete downstream tasks such as sizing and defect inspection.

Industry Case: Texture Dependence in Logistics Sorting

Logistics sorting lines often use stereo cameras to measure parcel volume (length, width, height) online, for billing and packing planning. In practice success or failure depends almost entirely on surface texture: a kraft cardboard box has printing, tape, and fiber texture on its surface, the disparity is dense and reliable, and the volume is measured fast and accurately; but the moment you encounter a black plastic bag, reflective sealing tape, or specular packaging, the surface is either textureless or the specular reflection breaks the Lambertian assumption, and the disparity fails over large areas — the measured volume is either incomplete or jumps around. There are two countermeasures: one is to switch to **active stereo**, adding projected speckle to artificially create texture on the surface (echoing Chapter 32); the other is multimodal hole-filling, using time-of-flight or structured light to patch where stereo fails. The lesson is direct: the prerequisite of passive stereo is that "the measured surface carries texture of its own," so before going live always verify the texture characteristics of the target surface, then decide whether to use passive stereo, active stereo, or switch to a different technology route.

31.7 Summary

- **Disparity is depth**: epipolar geometry constrains the correspondence search to one epipolar line, which degenerates after rectification into a one-dimensional in-row search; depth is given by $Z = fB/d$, and disparity is inversely proportional to depth.
- **Depth resolution worsens with Z^2** ($\Delta Z = Z^2/(fB)$): in this example 27.8 mm/px at 1000 mm, and being less accurate farther away is an intrinsic limitation of stereo, mitigated by increasing fB or by subpixel.
- **Block matching + winner-take-all** gives integer disparity (11×11 SAD: MAE 0.362 px), but integer quantization produces staircase banding; **parabolic subpixel interpolation** smooths the banding almost for free, dropping the ramp-region MAE from 0.252 to 0.070 px and raising the effective depth resolution from 27.8 mm to 2.78

mm — subpixel is the watershed of stereo doing metric measurement.

- **The left-right consistency check** honestly culls occlusion and textureless regions (valid rate 91.8% → 88.6%, but retained-pixel MAE improves to 0.134 px); **no texture, no stereo** — the textureless patch is 100% valid raw yet 82% bad, and still 82.5% bad after checking, which is exactly the motivation of active texture-projection schemes such as structured light.
- **Window size is a dilemma between accuracy and edge fidelity**: a large window is more accurate overall (MAE 0.251) yet makes the foreground edge bleed and swell (edge bad pixels 3.95% → 5.17%), with the box edge expanding outward by about half a window width per step, so engineering must compromise per scene or switch to adaptive/global methods.

For the classic treatment of epipolar and multi-view geometry, see the monograph by Hartley and Zisserman (Hartley and Zisserman 2004); the systematic taxonomy and evaluation of dense stereo matching costs and algorithms is given in the classic survey by Scharstein and Szeliski (Scharstein and Szeliski 2002), and semi-global matching (SGM), the industrial workhorse balancing accuracy and efficiency, was introduced by Hirschmüller (Hirschmüller 2008). The engineering realization and uncertainty analysis of stereo matching in industrial vision can be further consulted in the work of Steger et al. (Steger, Ulrich, and Wiedemann 2018).

32 Structured Light 3D Imaging

The Achilles' heel of binocular stereo vision (Chapter 31) is "finding correspondences": the left and right images must be matched at the same physical points, and that matching relies on surface texture. Yet the industrial floor is full of texture-poor objects—polished metal surfaces, the uniform painted coats of injection-molded parts, white ceramics—on which every pixel looks identical, leaving the matching algorithm with nothing to grab onto and the disparity map riddled with large empty holes. If the object will not grow its own texture, then we **project it on** actively. This is exactly the core idea of structured light: a projector casts a set of known, controllable bright-and-dark patterns onto the object's surface, making them serve as "artificial texture." When the pattern lands on an undulating surface, it deforms with the height; the camera captures that deformation and inverts it to recover the height at every point. Among all the codable patterns, **sinusoidal fringes + phase shifting** is the workhorse of industrial 3D metrology—it encodes the height into the **phase** of the fringes, and phase can deliver subpixel, per-pixel dense reconstruction. Figure 32.1 shows the set of four-step phase-shifted fringes that runs through this entire chapter.

32.1 Principle of Phase Shifting

Suppose the projector casts spatial sinusoidal fringes; the gray value the camera samples at pixel (x, y) can be written as

Why sinusoidal fringes rather than binary Gray code? Gray code encodes only "black/white" per pixel, so its resolution is limited by the finest line width of the fringes; the sinusoidal phase is a **continuous quantity**, and combined with phase shifting it can resolve the fractional phase inside a pixel, raising accuracy by one or two orders of magnitude. The price is that the phase has a 2π periodic ambiguity that must be unwrapped—precisely the problem Section 32.2 of this chapter sets out to solve.

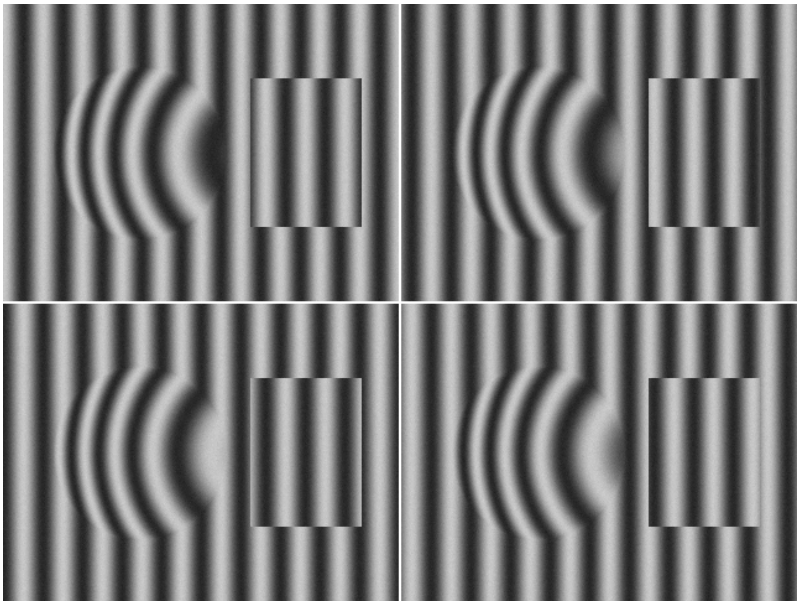


Figure 32.1: Four-step phase-shifted sinusoidal fringes (period $P = 32$ px), with a $\pi/2$ phase shift between adjacent frames. The scene is a flat base plane onto which a parabolic spherical cap (center) and a rectangular step (right) are superimposed; look closely and the fringes bend and shift laterally over the cap and the step—this is the direct evidence of height modulation.

$$I_k(x,y) = A(x,y) + B(x,y)\cos\big(\varphi(x,y) + k\,\tfrac{\pi}{2}\big), \qquad k = 0,1,2,3.$$

Here A is the background light (the DC component), B is the fringe **modulation**, i.e. the amplitude of the sinusoid, and φ is the **phase** we are really after—it encodes the height at that point. We project four fringe images in succession, each shifting the initial phase as a whole by $\pi/2$ (this is "four-step phase shifting"), so the same pixel yields four equations, and the unknowns are exactly the three quantities A, B, and φ. Subtracting the four equations pairwise:

$$I_3 - I_1 = 2B\sin\varphi, \qquad I_0 - I_2 = 2B\cos\varphi.$$

Dividing and taking the arctangent, A and B are eliminated together, leaving only the phase:

$$\varphi(x,y) = \operatorname{atan2}\,(I_3 - I_1,\; I_0 - I_2).$$

This formula is the whole essence of phase shifting. What is worth savoring repeatedly is the **source of its robustness**: φ depends only on the differences and ratio of the four intensities, and is independent of the background light A and of the fringe brightness B. In other words, camera gain drift, ambient-light fluctuation, and the high or low albedo of the measured surface none of them change the recovered phase—they alter only A and B, and both of those have already been eliminated. This is the same engineering wisdom as the "gradient direction does not change with illumination gain" of Chapter 17: **encode the information into a quantity that is invariant to gain, and the measurement becomes stable.** Intensity itself is fragile; phase is reliable.

The range of atan2 is $[-\pi, \pi)$, so the recovered phase is the **wrapped phase**—every time the true phase exceeds a 2π, it is folded back into this interval, forming a sawtooth.

Figure 32.2 illustrates this: a continuously undulating height field turns into bands of equal phase in the wrapped-phase image, with "cliffs" plunging from $+\pi$ down to $-\pi$ between the bands. These cliffs are not real discontinuities but artifacts of the periodicity of atan2; the next section will smooth them out.

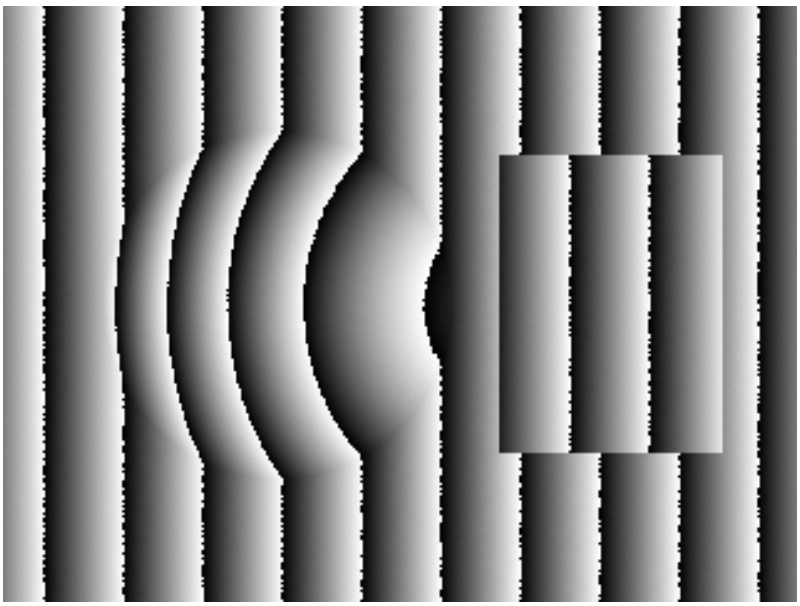


Figure 32.2: Wrapped phase $\varphi = \operatorname{atan2}(I_3 - I_1,\, I_0 - I_2)$, gray-mapped over $[-\pi, \pi)$. The 32 px fringe period forms about ten equal-phase bands across the frame, and the end of each band jumps abruptly from white ($+\pi$) to black ($-\pi$)—that is the wrapping cliff; the bending of the fringe bands over the cap and the step is the height information.

32.2 Phase Unwrapping

To restore the wrapped phase $\varphi \in [-\pi, \pi)$ to the continuous **absolute phase** $\Phi = \varphi + 2\pi n$, one must determine the integer **fringe order** n pixel by pixel. The most naive approach is **spatial unwrapping**: scan along the image, and whenever the phase difference between adjacent pixels

suddenly jumps past π, judge that a wrapping cliff has been crossed and add a 2π to all subsequent pixels. It works on smooth, connected surfaces, but it is disturbingly fragile—the moment it meets a false jump caused by noise, or a real height discontinuity on the object (such as the step in this chapter), the integration goes **wrong by one or more whole** 2π **from some point onward**, and the error propagates all the way along the scan direction with no way to self-heal.

More robust in industry is the **dual-/multi-frequency hierarchical method (dual-/multi-frequency unwrapping)**. Its idea is to use two sets of fringes with different periods: one **low-frequency** set with a period large enough that the phase varies by less than 2π across the entire field of view, so it is naturally unambiguous and can directly tell us "roughly which fringe we are on"; and another **high-frequency** set with a small period and a steep phase slope, providing the final measurement accuracy. Figure 32.3 is the low-frequency set used in this chapter ($P = 128$ px, four times the high-frequency $P = 32$). Specifically, we first perform one spatial unwrapping of the low-frequency wrapped phase to obtain the low-frequency absolute phase Φ_{lo} (the low-frequency field is smooth, the phase jump caused by the step is only about 0.6 rad, far less than π, so the spatial unwrapping will not go wrong), then use the frequency ratio $P_{\text{lo}}/P_{\text{hi}} = 4$ to scale it up and **predict** the high-frequency absolute phase, rounding to fix the order pixel by pixel:

$$\Phi_{\text{hi}} = \varphi_{\text{hi}} + 2\pi \,\text{round}\Big(\frac{\Phi_{\text{lo}} \cdot (P_{\text{lo}}/P_{\text{hi}}) - \varphi_{\text{hi}}}{2\pi}\Big).$$

The key point is this: the order n is computed **independently per pixel**, without relying on neighborhood integration, so an error at one pixel can never infect its neighbors—the real discontinuity of the step is preserved faithfully, rather than being "smoothed away." This is the fundamental reason the dual-frequency method dominates the spatial method. Figure 32.4 is the absolute phase after unwrapping: smooth and monotone, with the dome of the spherical cap and the block of the step clearly distinguishable.

Unwrapping has another technical route: **temporal phase unwrapping**. Instead of looking for cliffs in the spatial neighborhood, it projects, onto the **same pixel**, several sets of fringes of increasing frequency over time, using the absolute phase of the previous frequency to assign the order for the next. The dual-frequency method is its most stripped-down two-level special case. The temporal method's

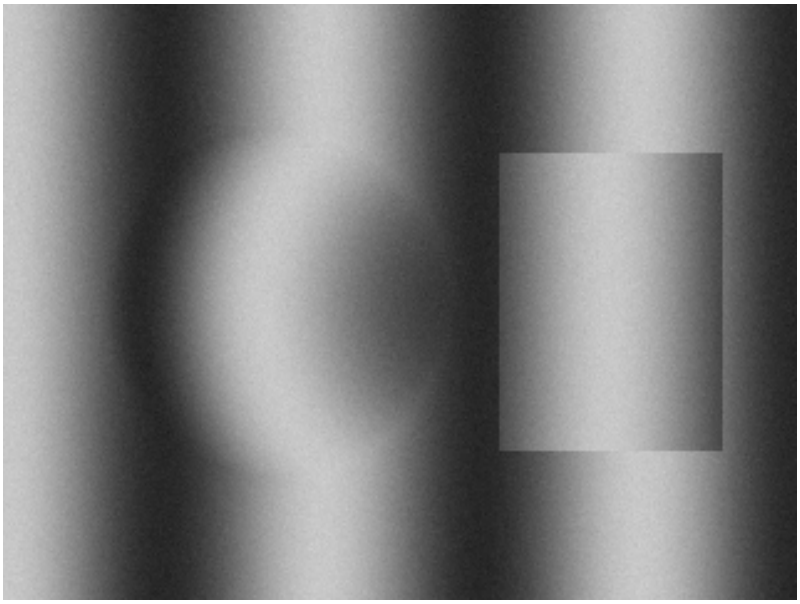


Figure 32.3: Low-frequency guide fringes ($P = 128$ px). The phase varies by less than a single 2π across the entire field of view, so there is no periodic ambiguity, and it can directly assign orders for the high-frequency fringes.

Figure 32.4: Absolute phase after dual-frequency hierarchical unwrapping. All wrapping cliffs are smoothed out, the phase rises continuously and monotonically with height, and the spherical cap (the central dome) and the step (the block on the right) are obvious at a glance.

32.3 Height Reconstruction and Accuracy

The final leap from absolute phase to height is a triangulation calibration relation. In this chapter's imaging model, the projector coordinate $u = x + c \cdot h$, where h is the height and c is the **triangulation constant** determined by the projector–camera geometry (here $c = 4$ px/mm, i.e. 1 mm of height shifts the fringe laterally by 4 px). The relation between the absolute phase and the projector coordinate is $\Phi = 2\pi u / P_{\text{hi}}$, and solving the two together inverts to:

$$h(x, y) = \frac{1}{c}\Big(\frac{\Phi_{\text{hi}}\, P_{\text{hi}}}{2\pi} - x\Big).$$

The measured accuracy is satisfying. Under the conditions $A = 120$, $B = 80$, noise $\sigma = 3$, the **apex of the spherical cap reconstructs to 5.994 mm, against a ground truth of 5.994 mm**; the **step height is 2.999 mm, against a ground truth of 3.000 mm**—both hit the mark. Figure 32.5 is the final height map. The RMS noise floor on the flat base plane is **0.0338 mm**, and that is the height repeatability of this configuration.

Where does this noise floor come from, and what determines it? The answer is the lifeblood of structured light—the **modulation** B. Propagating the error through four-step phase shifting, the standard deviation of the phase satisfies

$$\sigma_{\varphi} \approx \frac{\sigma}{B},$$

that is, the phase noise is inversely proportional to the fringe amplitude. Drop B from 80 to 15 (about 5.3×), and in theory the phase and height noise should be amplified by the same factor. Experiment shows: at $B = 15$ the base-plane RMS rises to **0.1823 mm**, exactly **5.4 times** the 0.0338 mm at $B = 80$—almost perfectly matching the 5.3-fold theoretical prediction. This $\sigma_{\varphi} \propto \sigma / B$ law translates an abstract image-quality metric (how clear the fringes are) directly into

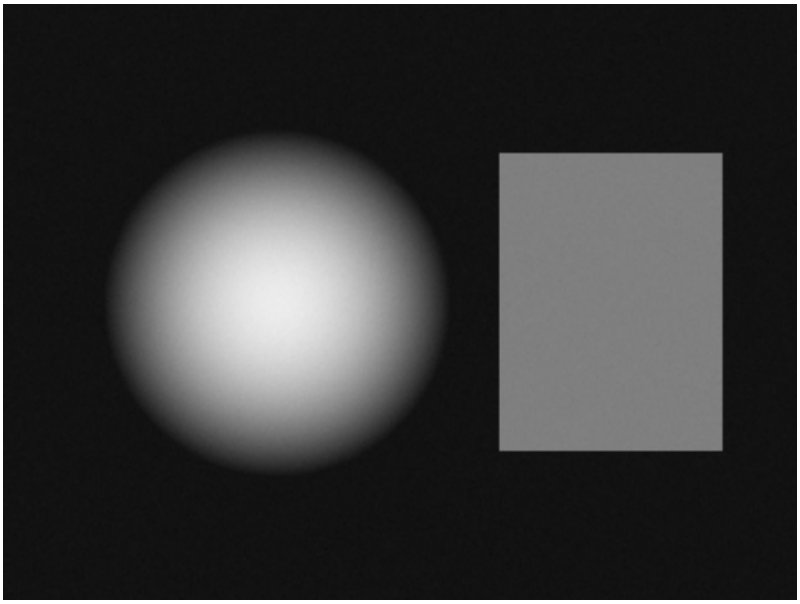


Figure 32.5: Reconstructed height map, gray-mapped over $[-0.5, 6.5]$ mm. The flat base plane is uniform, the spherical cap appears as a smooth dome, and the step is the bright block on the right—the 3D shape has been recovered densely, pixel by pixel.

height accuracy. **The accuracy of structured light is, in essence, determined by the modulation.** Every factor that suppresses B—dark surfaces, low-contrast projection, defocus, motion blur—degrades the 3D accuracy proportionally.

32.4 Modulation, Saturation, and Surfaces

Since B determines accuracy, a natural engineering move is to use a **modulation mask** to reject pixels whose B is too low—four-step phase shifting incidentally computes $B = \frac{1}{2}\sqrt{(I_3 - I_1)^2 + (I_0 - I_2)^2}$, and thresholding it does the job. In this chapter's "real-capture" experiment, a dark region with an albedo of only 0.18 is embedded in the lower-left corner of the base plane, simulating black paint or a low-reflectance surface. The result is beautiful: with $B < 20$ as the threshold, **99.7% of the dark-region pixels are flagged**, while the clean base plane has **zero false alarms**—the mask precisely circles out the untrustworthy region. The left of Figure 32.6 is the modulation map, the right is the binary mask.

But the modulation mask has one fatal blind spot: **it cannot block saturation.** The experiment overlays a specular highlight on top of the spherical cap (clipped at 255 after a 1.7× gain), simulating a metallic reflection. Intuitively, saturation would destroy the sinusoid and lower the modulation, so the mask ought to catch it—but the measured fraction of the **saturated region that is flagged is 0%**. The reason is this: the four phase-shifted images are clipped equally to 255, and the differences $I_3 - I_1$ and $I_0 - I_2$ still retain a fair amount of amplitude after clipping, so the computed B does not drop below the threshold. The mask therefore believes the region has "good modulation, trustworthy data," when in fact the phase has already been distorted by clipping. The consequence is that the apex of the spherical cap shifts from 5.994 mm to **5.928 mm**, the RMS inside the saturated disk rises to 0.1087 mm (about 3.2 times the clean region), and the dark-region RMS is 0.1911 mm.

$\sigma_\varphi \approx \sigma / B$ also explains why industrial structured-light systems strive to project **high-contrast** fringes: projector brightness, lens aperture, camera exposure—they all ultimately serve one goal, pushing B to its maximum without saturating. But the premise "without saturating" is crucial; the next section will show that once the line is crossed, catastrophe arrives in another form.

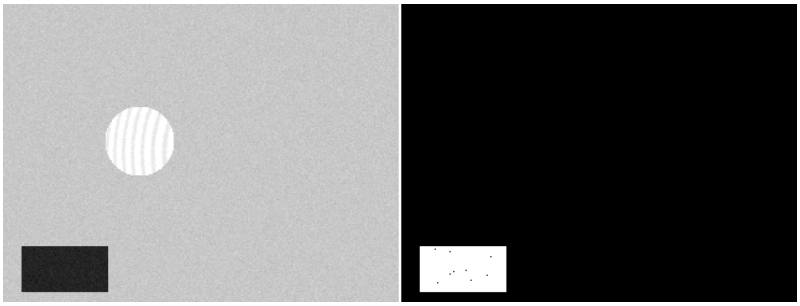


Figure 32.6: Left: modulation map, brightness proportional to B, with the dark region and the saturated highlight disk clearly visible; right: the binary mask for $B < 20$, where the dark region is flagged as a solid patch (white), while the saturated highlight disk on the cap is almost entirely unflagged—the mask is sensitive to the dark region yet blind to saturation.

The mask fails, and only the error map can expose the truth—Figure 32.7 plots the reconstruction error, and the error gathers into exactly two clusters: the dark region and that saturated highlight disk, precisely coinciding with the locations the mask missed.

This puts on the table the "compatibility" of different surfaces with structured light. Dark/low-albedo surfaces suppress B and amplify noise, and can be rejected by the mask; saturated highlights quietly inject a systematic bias right under the mask's nose and must be prevented at the source by exposure control (avoiding crossing the line; echoing the illumination design of Chapter 4); translucency and subsurface scattering make the fringes "bleed" and distort the phase. There is also a class of surfaces structured light simply cannot handle—**mirrors**: the light is reflected away by the mirror surface, and the camera never receives the fringes landing on the surface; here one must switch to deflectometry (Chapter 35), which measures not the fringe position but the deflection of the pattern by the surface normal. **No single 3D method works on every surface**; the boundary of structured light is precisely the boundary of its method assumptions (Chapter 30).

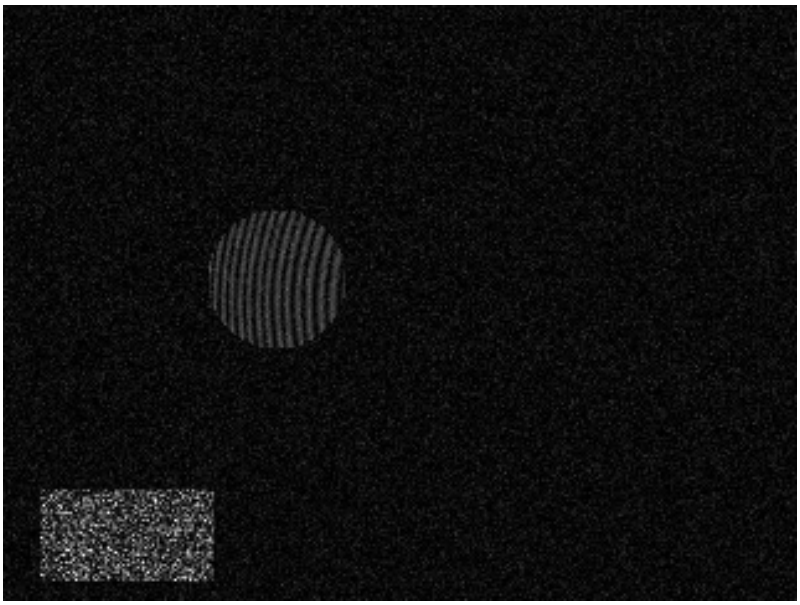


Figure 32.7: Height error map ($|h - h_{\mathrm{GT}}|$, gray over $[0, 0.6]$ mm). The error concentrates into two patches: the lower-left dark region and the central saturated highlight disk. The latter is precisely the region the modulation mask failed to flag—the mask cannot block saturation, so only the error map can expose it.

32.5 SciVision Implementation

This chapter originally planned to reuse the SDK's phase-measurement module `SCIMV::SciSvPhaseMeasure`, but its core entry points proved unusable in testing, faithfully recorded here. `DecodePatterns` (four-step phase-shift decoding), under all **16 call variants** we tried—8U and 32F input, with and without a prior `InitPMD`, UNDEF and full-image rectangular ROI, with and without a supplementary set of Y-direction fringes—**all returned rc=0 yet produced no output whatsoever**: the phase map, gray map, and modulation map were all 0×0 empty objects, and the DLL even printed a "Directory does not exist." to stderr (apparently a missing runtime resource directory). `CalUnPhase` (spatial unwrapping), which depends on its output, was consequently unusable as well. The entire pipeline is therefore **written entirely by hand**. There are only two core snippets, but they are the whole of this chapter's algorithm:

```
// 1. Four-step phase-shift wrapped phase + modulation (A, B cancel in the differences, leavin
for (size_t i = 0; i < (size_t)W * H; ++i) {
    double s = (double)im[3][i] - im[1][i];   // =  2B*sin(phi)
    double c = (double)im[0][i] - im[2][i];   // =  2B*cos(phi)
    phi[i] = std::atan2(s, c);                // wrapped phase [-pi, pi)
    mod[i] = 0.5 * std::sqrt(s * s + c * c);   // modulation ~ B
}

// 2. Dual-frequency hierarchical unwrapping: low-freq absolute phase *4 predicts high-freq or
const double ratio = P_LO / P_HI;             // 128/32 = 4
for (size_t i = 0; i < abs.size(); ++i)
    abs[i] = wHi[i] + TWO_PI * std::round((uLo[i] * ratio - wHi[i]) / TWO_PI);
```

The first snippet directly implements the arctangent formula of Section 32.1: `im[k]` is the k-th phase-shifted image, `s` and `c` are proportional to $\sin\varphi$ and $\cos\varphi$ respectively, `atan2` gives the wrapped phase, and the modulation is obtained incidentally from the magnitude of the two. The second snippet is the hierarchical method of Section 32.2: `uLo` is the spatially unwrapped low-frequency absolute phase, which

after multiplying by the frequency ratio 4 predicts the high-frequency absolute phase, and `std::round` assigns each pixel independently to the nearest integer multiple of 2π. The complete runnable project is at `code/structured_light/`, including fringe synthesis, the low-modulation rerun, and the real-capture trap experiments.

Industry Case: Reflective Solder and Black Ink in SPI

PCB solder-paste printing inspection (SPI, Solder Paste Inspection) is a classic battlefield for structured-light 3D—it must measure the volume of every blob of solder paste point by point, directly bearing on the soldering yield. The difficulty is exactly this chapter's dilemma in one frame: the freshly printed **solder paste** has a metallic luster and is highly prone to specular reflection and saturation; the **black solder-mask ink** beside it has extremely low albedo and all but swallows the light. Tune the exposure for the solder paste, and the black-ink region's phase drowns in noise; tune it for the black ink, and the solder-paste highlights saturate and inject a systematic height bias—and saturation, in turn, fools the modulation mask, so the bias creeps in silently.

The practical countermeasure is **HDR multi-exposure fused fringes** (echoing Chapter 12): for each phase-shift phase, project several exposure levels, and per pixel pick the level that is unsaturated and has the highest modulation to fuse, letting the solder paste and the black ink each take what they need. The lesson is plain yet profound: **the whole-field accuracy of structured light is determined by the worst modulation anywhere in the field**—optimization must target the weakest surface, not the average one.

32.6 Summary

- **No texture? Project it on.** Binocular stereo relies on surface texture to find correspondences, and smooth objects make it fail; structured light projects a controllable "artificial texture," encoding height into the fringe phase to achieve dense 3D reconstruction.

- **Phase is more robust than intensity.** Four-step phase shifting $\varphi = \text{atan2}(I_3 - I_1, I_0 - I_2)$ eliminates both the background light A and the fringe amplitude B, making the phase insensitive to gain, ambient light, and albedo—this is the foundation of stable measurement.
- **Wrapping must be undone; dual-frequency is the most robust.** The 2π ambiguity of atan2 must be removed by unwrapping; the spatial method integrates point by point and collapses at discontinuities, while the dual-/multi-frequency hierarchical method assigns orders per pixel and resists discontinuities, and is the industry standard.
- **Modulation is the lifeblood.** The height noise $\sigma_\varphi \propto \sigma/B$; measured, B dropping 5.3× raised the error 5.4×; every factor that suppresses B degrades accuracy proportionally.
- **The mask cannot block saturation.** The modulation mask can precisely reject the dark region (99.7%, zero false alarms), yet is blind to saturation (0% flagged)—saturation does not lower the modulation but distorts the phase, and only the error map can expose it; defending against saturation relies on exposure control and HDR, not an after-the-fact mask.

For a systematic taxonomy of structured-light coding patterns, see the survey by Salvi et al. (Salvi et al. 2010); the development and key issues of fringe-projection profilometry are reviewed by Gorthi and Rastogi (Gorthi and Rastogi 2010), while Geng's tutorial (Geng 2011) gives a systematic and accessible overview of the various structured-light 3D-imaging methods. Implementation details of phase-shift profilometry and multi-frequency unwrapping in industry can be further consulted in the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

33 Laser Triangulation

Structured light (Chapter 32) projects a whole sheet of coded texture onto an object and recovers the full-field height in a single exposure. Laser triangulation takes the opposite route: it casts out just **one** laser line, and a single exposure measures only the one profile where that line slices across the object. It seems like far less information, but paired with uniform motion of the object or the camera, stacking profile after profile along the scan direction reconstructs an entire 3D shape all the same — and the single-line scheme concentrates the light energy, keeps extraction simple, and runs blisteringly fast, making it the workhorse of online industrial 3D inspection: burr-height gauging on battery electrodes, weld reinforcement and undercut, profile wear on rails — behind most of these is a laser profilometer scanning the line at kilohertz profile rates. This chapter threads all of its experiments through one synthetic "profile target" scene (Figure 33.1): a flat conveyor belt carries a triangular prism and a circular-arc bump, a line laser strikes from above at a slant, and the camera looks down from another angle at the bright line that the objects lift up and bend.

33.1 The Triangulation Principle

What the laser emits is not a single beam but a **sheet** of light — a laser plane spread out by a cylindrical lens — which intersects the measured surface along a spatial curve. The camera observes this line from a direction making a fixed angle with the laser plane: when a surface point rises, the intersection shifts along the laser plane, which on the camera's image plane appears as a **row-direction displacement** of the laser line in that column. Between height and displacement there is a one-to-one geometric

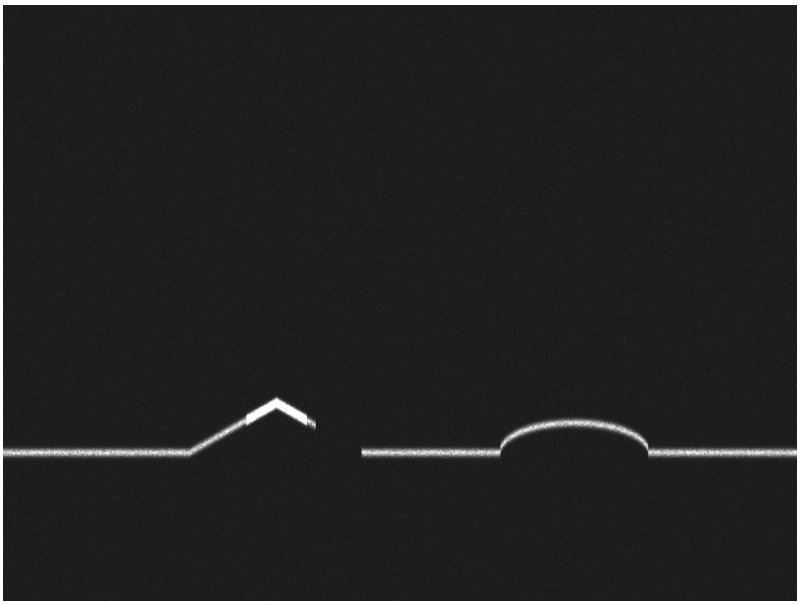


Figure 33.1: Line-laser camera view (640×480, the raw bright line before inverted display). The flat conveyor belt gives a horizontal baseline, the triangular prism folds the laser line into a sharp apex (a short top segment is overexposed by specular reflection and clipped into a plateau), and the circular-arc bump bends the line into a smooth bow; the prism's lower-right slope is shadowed from the laser by its own ridge, leaving a gap with no bright line.

relation — and this is exactly where the word "triangulation" comes from: the laser plane, the camera optical axis, and the measured point form a triangle, and given the baseline (the relative pose of camera and laser) and the observation angle, the depth of the third vertex can be solved.

After calibration, this geometry collapses into a single calibration constant. The synthetic data in this chapter takes the laser baseline row BASEROW = 360 (the laser row corresponding to the zero-height conveyor belt) and makes the row displacement proportional to the height, with proportionality coefficient $k = 0.8$ px/mm. So when the laser line in some column lands on row r, the height of that column is

$$h = \frac{\text{BASEROW} - r}{k}.$$

A real system's calibration curve need not be this linear, but the form is unchanged: **extracting the row position of the laser line is equivalent to measuring the height**. So the accuracy of laser triangulation, in the end, depends on a single thing — how precisely you can find the center of the laser line in each column.

Compared with structured light, laser triangulation is a "single line vs. full field" trade-off: structured light measures the full field in one exposure, but must project multiple coded patterns, unwrap phase, and guard against crosstalk; laser triangulation measures only one line per frame and trades scanning for area, but pushes the signal-to-noise ratio and extraction accuracy of each line to the extreme. Both belong to active triangulation (Chapter 30); they simply place "information content" and "acquisition speed" on opposite pans of the scale.

The slanted laser plane is not parallel to the camera image plane, so with ordinary focusing only a short segment of the laser line is sharp and both ends are defocused. The Scheimpflug principle makes the lens plane, the image plane, and the laser plane meet along a common line, so the entire laser line falls within the depth of field at once — this is why profilometer lenses are commonly mounted tilted.

33.2 Laser Line Extraction

Laser line extraction is a column-by-column, independent one-dimensional subpixel peak-localization problem: in each

column's gray profile, the laser line is a near-Gaussian bright peak, and what we want to estimate is the peak's **subpixel row coordinate**. This is the same idea as the edge subpixel interpolation of Chapter 14 and the caliper localization of Chapter 20 — break an expensive two-dimensional localization into a string of cheap one-dimensional estimates, then cash in accuracy far beyond the pixel grid with subpixel interpolation. The two mainstream algorithms follow.

Center of gravity (CoG) treats the laser peak as a stretch of "mass distribution" and takes its gray-weighted centroid as the line center. Let the gray value of row r be I_r and the background estimate be b; within a window near the peak,

$$\hat{r}_{\mathrm{CoG}} = \frac{\sum_r (I_r - b)\, r}{\sum_r (I_r - b)}.$$

It requires no model assumption and is extremely cheap to compute — the default implementation in industrial profilometers.

Gaussian fit assumes the laser cross section is a Gaussian peak $I_r - b \approx a \exp[-(r-\mu)^2/2\sigma^2]$. Taking the logarithm of both sides yields a quadratic in r, so one does a weighted parabolic fit of $\ln(I_r - b)$ as $\ln(I_r - b) = A + B(r - c_0) + C(r - c_0)^2$, and the peak position is given by the vertex

$$\hat{r}_{\mathrm{fit}} = c_0 - \frac{B}{2C}.$$

The fit adds a layer of model prior and is in theory more robust to noise, at the cost of more computation and a dependence on the assumption that "the peak really is Gaussian."

Intuitively the Gaussian fit is "more sophisticated," yet the measurements give a counterintuitive result. Comparing against ground truth column by column over the normal segment of Figure 33.1 (the unsaturated, unoccluded

The synthetic laser cross section in this chapter is Gaussian ($\sigma = 2$ px, peak 220, background 30), with **multiplicative** speckle noise added — the random interference spots produced when coherent laser light hits a rough surface, whose intensity is proportional to the signal itself. This point is decisive for the contest between CoG and the fit below: multiplicative noise is strongest at the peak top, exactly the samples the fit weighs most.

columns): **CoG has a mean |error| = 0.072 px, the Gaussian fit** 0.111 **px** — the center of gravity actually wins by a hair. The reason lies precisely in speckle's multiplicative character: the fit's log transform amplifies the weight of weak-signal samples and also weighs the high-gray points near the peak top most heavily, and that is exactly where multiplicative speckle is fiercest; CoG takes a weighted average over the whole window and averages the random spots away, ending up more stable. This is an honest detail: **there is no universally optimal extraction algorithm** — it depends on the statistical structure of the noise. The next section shows that the fit's real value only emerges when CoG fails outright.

33.3 Saturation: CoG's Achilles Heel

Engineers often assume "the brighter the laser the better," because brightness means a high signal-to-noise ratio. But gray values have a ceiling: an 8-bit sensor clips at 255. When the laser peak is clipped flat, disaster follows. Figure 33.2 compares two columns' cross sections: the normal column is a complete, symmetric Gaussian peak, while the column at the prism apex, overexposed by specular reflection, has its peak top sheared off into a flat plateau — the symmetry information carried by the peak shape has been cut away.

CoG is systematically biased on a clipped profile. It centers a window on the maximum pixel and takes the weighted average — but the plateau degenerates the "maximum pixel" into a whole stretch of tied highest points, and the centroid is dragged toward one side of the plateau, so the entire extracted line is pulled downward. The column-by-column statistics are striking: over the 49-column saturated segment, **CoG has a mean |error| = 0.387 px and bias** -0.387 **px** — 5.4 times the normal segment, and entirely one-directional bias (not random jitter but a systematic low reading). Converted to height, the CoG estimate at the prism apex **overestimates by** $+0.585$ **mm** relative to the truth. The Gaussian fit, by contrast, **discards the clipped samples with** $I \geq 250$ during fitting and uses only the unsaturated

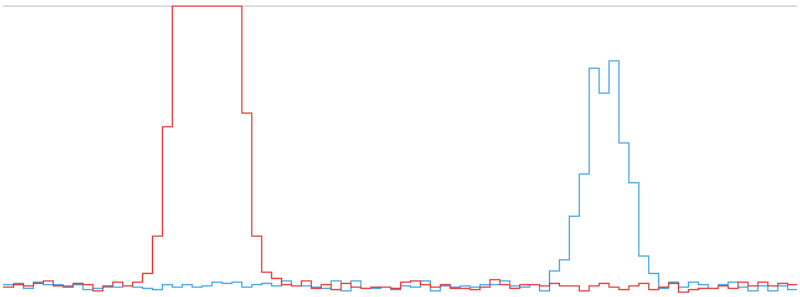


Figure 33.2: Comparison of two laser cross sections (horizontal axis: row coordinate magnified 8×; vertical axis: gray value). Blue is the complete Gaussian peak of the normal segment; red is the saturated column at the prism apex, with its top clipped into a plateau at 255 and the symmetric peak-tip information lost.

"Gaussian shoulders" on either side of the plateau to back out the peak position: the saturated segment's mean |error| drops to 0.078 px with a bias of just +0.005 px, and the height error at the prism apex is −0.046 mm. Figure 33.3 draws this reversal clearly: inside the saturation band (yellow), CoG (red) collapses into the negative-error region, while the fit (blue) stays glued to the zero-error line throughout.

There is a reversal here that must be reported honestly. In the previous section's normal segment, CoG (0.072) was still slightly better than the fit (0.111); the fit **wins only when clipping destroys CoG**. In other words, the fit is not "the more accurate algorithm" but "the algorithm more robust to saturation." The lesson has two layers: first, **the foremost red line of exposure is never to let the laser peak saturate** — overexposure brings not a higher SNR but a systematic bias that cannot be repaired after the fact; second, **when the profile may saturate, CoG silently overestimates the height**, and for specular, glossy-painted, and other easily reflective surfaces one must switch to a fit that discards clipped samples, or flag the saturated columns separately.

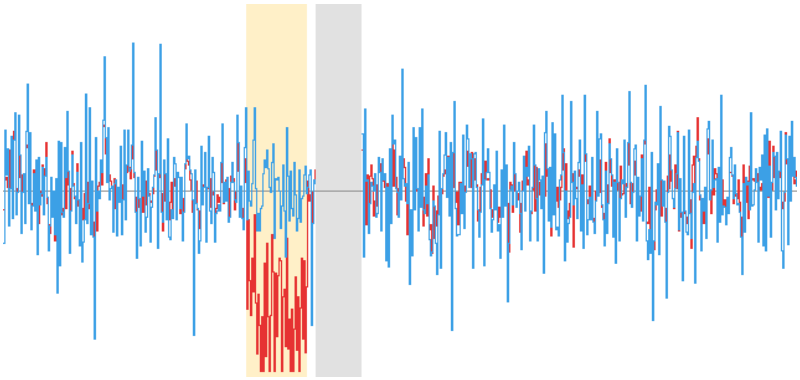


Figure 33.3: Per-column extraction error (red CoG, blue Gaussian fit; the yellow band is the saturated segment, the gray band the occlusion gap). Over the normal segment both hug the zero-error line, with CoG slightly less noisy; once inside the saturation band CoG develops a pronounced one-directional negative bias (the laser line is pulled downward), while the fit is barely affected.

33.4 Scanning and the Range Image

Feeding each column's extracted laser-line row position into $h = (\text{BASEROW} - r)/k$ turns one frame into a single height profile. Figure 33.4 overlays this reconstructed profile (black dots) on the ground truth (gray line): the triangular prism is a sharp-apex triangle and the circular-arc bump is a bow, both hugging the truth precisely; while over the segment shadowed by the ridge, the reconstructed profile is **simply left blank** — no laser means no data, and we never connect the gap with a fabricated curve.

A single profile is only one slice. Let the object pass uniformly through the laser plane on the conveyor belt and let the camera shoot continuously at a fixed frame rate, extracting one profile per frame and stacking them in order along the scan direction (rows), and you obtain a **range image** — where each pixel's "gray value" is no longer brightness but the height at that location. It is a 2.5D representation between a two-dimensional image and a full three-dimensional point

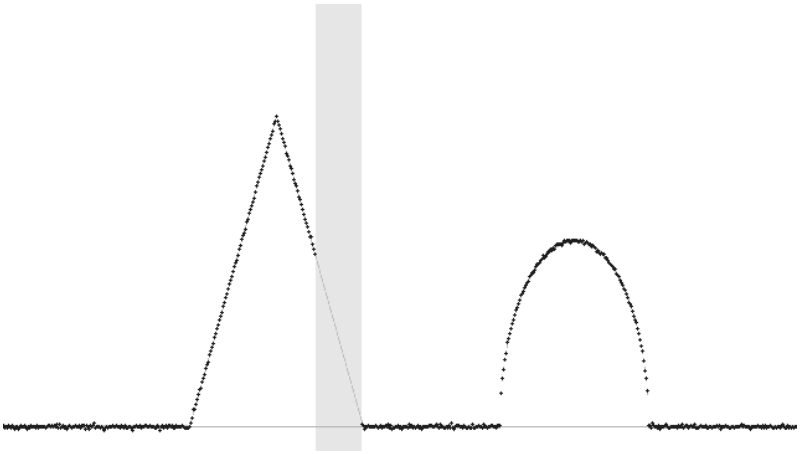


Figure 33.4: Single-frame reconstructed height profile (black dots are the extraction result, the gray line is ground truth; the gray band is the occlusion gap). The reconstructed profiles of the triangular prism (truth 50 mm) and the circular-arc bump (truth 30 mm) hug the truth tightly; the occluded segment is left blank, not filled by interpolation.

cloud: it has a regular pixel grid, but each cell carries a z value. Figure 33.5 is the range image stitched from this chapter's 200 scan lines $\times$ 640 columns, brighter for higher: the prism is a bright ridge with a slope, the circular-arc bump a soft bright spot.

Note that **pure-black vertical band** to the right of the prism: the prism's lower slope is blocked by its own ridge, the laser cannot reach it at all, and so the camera sees no bright line. In this frame 37 columns detect no laser peak, and the extraction algorithm faithfully flags all 37 columns as invalid (37/37), leaving an **honest hole** in the range image, never filled by interpolation. This is the exact opposite of the "silent failure" in Chapter 26: better to leave a visible hole than to fabricate a height whose falsity cannot be seen. Occlusion is a physical fact, and a fabricated smooth surface would lead downstream to misjudge a surface that simply does not exist. The whole range image has 3700 invalid pixels and a peak height of 50.268 mm, all concentrated in the shadow region. The two profiles' measurements also stand up to scrutiny: the

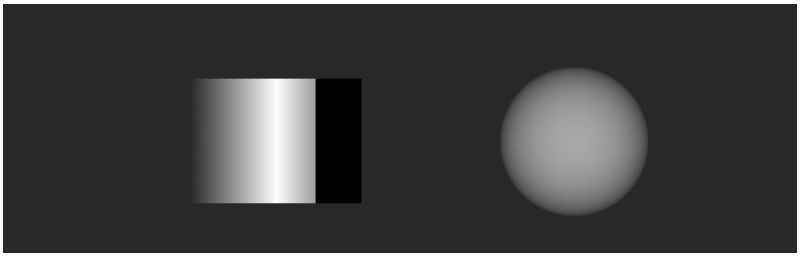


Figure 33.5: The scan-stitched range image (200×640, height encoded as gray value). The triangular prism is a bright region with a slope, and the pure-black vertical band to its right is the shadow gap occluded by the ridge (invalid pixels, not filled by interpolation); the circular-arc bump is the round bright spot on the right.

prism measures 49.954 mm (truth 50) and the circular-arc bump 29.918 mm (truth 30), agreeing at the submillimeter level.

The range image is the bridge to point cloud processing in Part IX of this book: give each valid pixel its physical (x, y) coordinate and z height, and the range image unfolds directly into an organized point cloud, with subsequent registration, segmentation, and measurement developed in Chapter 37.

SciVision carries the range image with `SciRangeImage`: underneath are `ushort` raw counts, paired with three-axis resolutions `resolutionX/Y/Z` and an offset `offsetZ`; `GetValue` returns the **raw count** (not the z-scaled value), so the physical height must be obtained by multiplying by `resolutionZ` and adding `offsetZ` yourself; `Save` outputs a 16-bit PNG to preserve height precision. In this example `GetValue` reads 49.940 mm at the prism apex. It belongs, alongside `SciPointCloud` and `SciTriangleMesh`, to the core 3D data types of `Sci3DData`.

33.5 Repeatability and Exposure

How good a height-gauging system is depends not only on how accurate a single measurement is, but more on the **scatter of repeated measurements of the same spot** — exactly the repeatability methodology emphasized in Chapter 20. We take 100 columns in a purely flat region, fix the geometry and vary only the random noise seed, repeatedly render and extract 50 times, compute the standard deviation of each column's extracted position, and then average over all columns.

Laser brightness (exposure) is a double-edged sword here. At peak 220 (normal exposure), CoG's standard deviation is 0.091 px (equivalent to 0.114 mm) and the fit's 0.139 px;

dimming the laser to peak 100 raises CoG to 0.112 px (0.140 mm) and the fit to 0.163 px. The pattern is clear: **the dimmer the laser, the lower the SNR, the worse the repeatability** — underexposure is the source of random error. But the previous section told us that overexposure (saturation) brings a far more frightening systematic bias. Squeezed from both ends, exposure can only take a compromise: bright enough for sufficient SNR, yet not letting any column saturate. In practice one often tunes to "the highest real surface just barely not clipping" and keeps a saturation flag on anomalously reflective columns as a backstop.

33.6 SciVision Implementation

An SDK defect must be recorded faithfully. This book's 3D laser-triangulation module `SciSv3DLaserTriangle` (v26.1) **crashes with an access violation (0xC0000005) at every entry point** on this machine: `FindLaserLine` (both overloads, every parameter combination), `AnalysisLaserline`, and `ReconstructeImage` — 20/20 crashes, regardless of whether fed a clean noise-free image, 8U or 16U, or any ROI (including the default UNDEF). So as not to let one crash drag down the whole example, the project isolates the SDK calls into a **subprocess probe** — the main process calls `system("demo.exe sdkprobe")` to spawn a child that pokes at the SDK, and a crash only makes the child exit abnormally, captured and logged by the main process; once some SDK version fixes it, the comparison result will automatically reappear. Laser line extraction is then done entirely with hand-written CoG and Gaussian fit (the source of all the numbers above).

```
// Subprocess probe: try the SDK entry (20/20 crashes here, kept as a baseline once the SDK is
SCIMV::SciSv3DLaserTriangle lt;
SciPointArray pts;  SciVarArray grays;
long rc = lt.FindLaserLine(img, roi, /*thresh*/60, /*width*/20, /*sigma*/3, &pts, &grays);
// rc!=0 records a failure; a child crash is caught by the return code of the main process's sy
```

On the range-image side, the construction and saving of `SciRangeImage` work fine and can be used with confidence:

```
// u16: raw count of each pixel's height (mm×100); resolution/offset are passed at constructio
SciRangeImage ri(u16.data(), W, NSCAN, W * (int)sizeof(unsigned short), SCI_DATA_USHORT,
                 1.0, 5002.0, /*resZ*/0.01, /*offZ*/-0.01, /*resX*/XRES, 0.0, /*resY*/YSTEP, 0
double v = ri.GetValue(100, 220);                  // raw count, not physical height
double z = v * ri.ResolutionZ() + ri.OffsetZ();    // → physical height mm
ri.Save("out\\range_image_sci.png");               // 16-bit PNG, preserves height precision
```

The expected interface of `FindLaserLine` is listed here as well, for reference once the SDK is fixed: input the laser image and the search ROI, give a peak-strength threshold, a laser line width, and a smoothing σ, and output a string of subpixel laser points `pts` with corresponding gray values `grays`; the hand-written extraction in this example implements exactly the column-by-column CoG/Gaussian fit according to this semantics. The complete project that generates all of this chapter's images and numbers lives in `code/laser_triangulation/`.

Industry Case: Burr-Height Inspection on Battery Electrodes

After lithium-battery electrode sheets are slit, burrs tens of microns high often remain along the edge; too tall a burr can pierce the separator and cause an internal short, so the burr height must be measured online and the part rejected. A line-laser profilometer scans across the electrode edge at kilohertz profile rates and should be more than capable of µm-level height gauging — but the electrode surface is mirror-like metal foil and coating, the laser striking it reflects strongly, and the peak easily shoots into saturation. Once clipped, CoG systematically overestimates the height by the mechanism revealed in this chapter: an edge that is not actually out of spec gets measured as a spuriously tall "burr," causing mass false rejection (overkill). There are three countermeasures: first, suppress the reflection in hardware — lower the laser power and add a polarizer to suppress the specular component, bringing the laser peak back into the unsaturated range; second, switch the extraction algorithm from CoG to a Gaussian fit that discards clipped samples;

third, flag the still-saturated columns separately and hand them to re-inspection rather than scrapping outright. The lesson is hard: for µm-level laser height gauging, the extraction algorithm matters as much as exposure control and the lens — even the most expensive profilometer, paired with an extraction algorithm that saturation biases, measures a false height.

33.7 Summary

- **Laser triangulation trades a single line for extreme single-profile accuracy and stitches a full surface by scanning**: the triangular geometry of the laser plane and the camera encodes height as the row displacement of the laser line ($h = (\text{BASEROW} - r)/k$); against structured light it is a "single line vs. full field, speed vs. information content" trade-off.
- **Extraction accuracy decides everything, but no algorithm is universal**: in the normal segment under multiplicative speckle, the center of gravity (CoG, 0.072 px) actually edges out the Gaussian fit (0.111 px) — the optimal extraction depends on the statistical structure of the noise and must be measured, not assumed.
- **Saturation is CoG's Achilles heel**: a clipped profile biases CoG systematically (saturated segment −0.387 px, height overestimated by +0.585 mm, 5.4 times the normal segment), while the Gaussian fit drops to 0.078 px after discarding clipped samples. The foremost red line of exposure is never to let the laser peak saturate.
- **Occlusion should leave an honest hole, never interpolate**: in the range image all 37 occluded columns are flagged invalid — better empty than fabricated, since a faked smooth surface is far more dangerous than a visible hole.
- **Exposure is a compromise between underexposure's random error and overexposure's systematic bias**: at peak 220 CoG's repeatability is 0.091 px, rising to 0.112 px when dimmed to 100; the tuning target is "the highest real surface just barely not clipping."

The theoretical foundation of subpixel laser-line extraction is the unbiased curvilinear-structure detector proposed by Steger (Steger 1998), which yields subpixel centerline localization together with width estimation; a fast and robust laser-stripe extraction method for the reflections and noise of industrial sites is given by Usamentiaga et al. (Usamentiaga, Molleda, and García 2012). A more systematic treatment of laser triangulation and 3D reconstruction can be further consulted in the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

34 Photometric Stereo

The three-dimensional imaging of the previous chapters, whether laser triangulation (Chapter 30) or deflectometry (Chapter 35), measured the **macroscopic height** of a surface — how far a part protrudes, how deep it is dented. But industrial inspection often cares about something else entirely: which **direction** each point of the surface actually faces. The sharpness of a stamped character, the texture orientation of a frosted surface, the slope of a shallow defect — at root these are all questions of **surface normal**, and the normal is insensitive to absolute height. **Photometric stereo** is made precisely for this: the camera stays fixed, and we simply switch among several light sources of known direction, capturing one image at a time; the same point appears bright or dark differently under different illumination, and this variation in brightness encodes exactly its orientation. By solving the multiple images jointly, we can recover at every pixel both its normal and its **albedo**.

This technique also has a unique trick that is hard to replace elsewhere: it can cleanly separate "shape defects" from "appearance defects" into two distinct images. A dent changes the normal; a stain changes the albedo — a single-illumination grayscale image mixes the two together, whereas photometric stereo decouples them naturally. This chapter runs all its experiments on one synthetic relief board: a flat base plane carrying a raised cross-shaped logo, a 0.3 mm shallow dent defect, and a stain that alters only the albedo, not the height — exactly the "pair that ought to be separated" in inspection. Figure 34.1 shows the inputs captured under four illumination directions (slant angle 45°, azimuth 0°/90°/180°/270°).

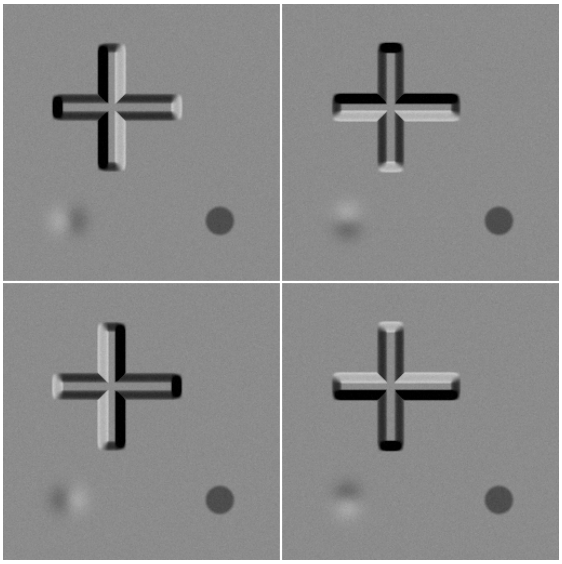


Figure 34.1: Images of the relief board captured under four lights of known direction (slant angle 45°, azimuth 0°/90°/180°/270°). The same raised logo flips its bright and dark sides under different illumination, while the stain disk in the upper left stays dark in all four images — the variation of brightness with illumination encodes the normal, and the constant darkening encodes the albedo.

34.1 The Lambertian Model and Solving for Normals

Photometric stereo is built on the **Lambertian reflectance** model (echoing the diffuse reflection of Chapter 4): the brightness of an ideal diffuse point depends only on the cosine of the angle between its normal $\mathbf{n}$ and the incident light direction $\mathbf{l}$, independent of the viewing direction:

$$I = \rho\,(\mathbf{n}\cdot\mathbf{l}), \qquad \mathbf{n}\cdot\mathbf{l} > 0,$$

where ρ is the albedo of that point (how much it absorbs, how much it reflects), and $\mathbf{l}$ is the unit lighting vector. Note that the unknowns come in two parts: the normal $\mathbf{n}$ (two degrees of freedom, since $\|\mathbf{n}\| = 1$) and the scalar ρ — three unknowns in all. A single image gives only one equation, far from enough; but if we capture one image under each of K light sources of **known direction**, we obtain K equations.

The key trick in solving is to combine ρ and $\mathbf{n}$ into a single vector $\mathbf{g} = \rho\,\mathbf{n}$, whereupon the equation becomes **linear** in the unknown: the k-th light source gives $I_k = \mathbf{g}\cdot\mathbf{l}_k$. Stacking the K equations into the matrix form $\mathbf{I} = L\,\mathbf{g}$ (each row of L is one light vector), $\mathbf{g}$ is uniquely determined when $K = 3$ and the three light directions are linearly independent; when $K \geq 4$ the system is overdetermined, and we solve the normal equations by least squares (echoing Chapter 2):

$$\mathbf{g} = (L^\top L)^{-1} L^\top \mathbf{I}.$$

Once $\mathbf{g}$ is solved, the decomposition is extremely clean — **the magnitude is the albedo, the direction is the normal**:

$$\rho = \|\mathbf{g}\|, \qquad \mathbf{n} = \mathbf{g}/\|\mathbf{g}\|.$$

Why are 4 light sources better than 3? With 3 sources the solution is exactly determined, and any noise, any slight non-Lambertian behavior, enters the result undiminished; with 4 sources the system becomes overdetermined, and least squares averages across four measurements so that noise is partly canceled. The experiments of this chapter use 4 sources, with a full-image mean normal error of about 1.9°, already approaching the noise floor of $\sigma = 3$.

This chapter solves this small 4×3 system pixel by pixel. Encoding the normal as color by $r = (n_x{+}1)/2$, $g = (n_y{+}1)/2$, $b = n_z$ yields Figure 34.2: the flat region appears blue because $\mathbf{n} \approx (0, 0, 1)$, while the four side walls of the logo face different directions and take on red and green tints; the albedo image Figure 34.3 strips away the orientation information entirely, leaving only "how strongly each point reflects" — here the stain disk is a clear dark spot, while the dent defect is almost invisible. These two images come from the same set of data yet carry only "shape" and "appearance" respectively, which is exactly the protagonist of the next section.

34.2 Separating Shape from Appearance

This is the **killer-app experiment** of photometric stereo. On the relief board we deliberately placed two utterly different kinds of defect: a 0.3 mm Gaussian dent (pure shape change, albedo unchanged) and a stain (pure albedo change, height unchanged). We take statistics over a small disk at each of three locations: a clean flat bottom as the baseline, the dent, and the stain. The results are in the table below, also corresponding to the left and right channels of Figure 34.4.

Location	Normal deviation from vertical	Albedo
Flat-bottom baseline	0.99°	0.901
Dent (0.3 mm)	**8.26°** (8.3× the baseline)	0.900 (−0.1%, unchanged)
Stain	1.92° (near noise floor)	**0.500** (55% of the baseline)

The numbers tell the story plainly: **the dent shows up only in the normal channel** — 8.26° against the 0.99° baseline is a jump of more than 8×, while its albedo does not budge; **the stain shows up only in the albedo channel** — the albedo drops to 55% of the baseline, while its normal deviation is only 1.92°, all but buried in the noise floor. Shape defects and

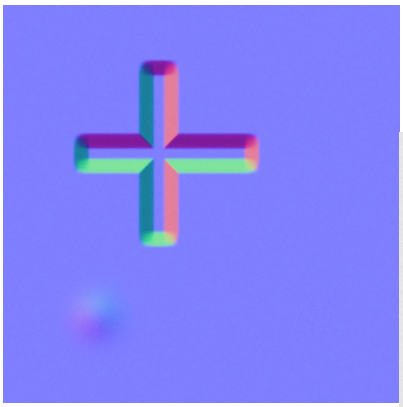


Figure 34.2: Normal map (RGB encoding: R=n_x, G=n_y, B=n_z). The flat base appears blue (normal facing the camera), the logo side walls take on red and green tints because they face different directions, and the stain region matches the base color — appearance defects do not enter the normal channel.

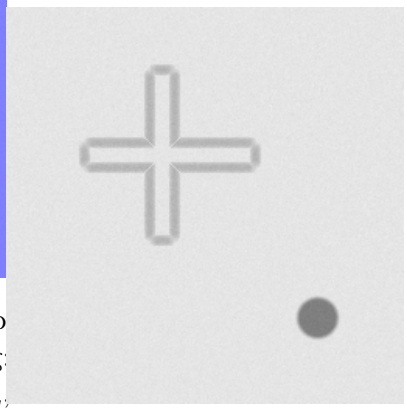


Figure 34.3: Albedo map. The stain disk is a clear dark spot (albedo about half the baseline), while the 0.3 mm dent defect is almost invisible here — shape defects do not enter the albedo channel.

appearance defects are sent by photometric stereo into two mutually non-interfering channels.

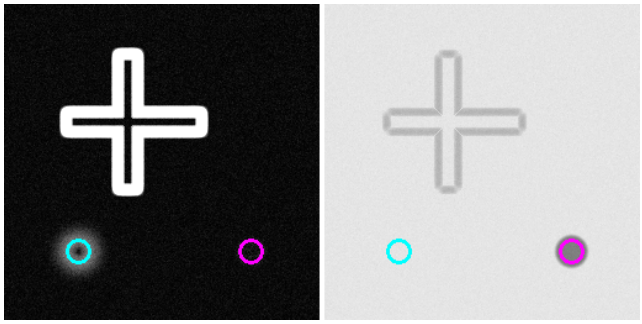


Figure 34.4: Separation experiment: left is the normal-deviation heat map, right is the albedo map, with both marking the dent (cyan ring) and the stain (magenta ring). The dent is bright on the left and invisible on the right; the stain is dark on the right and gives no response on the left — shape and appearance each occupy one channel.

Why can 2D grayscale not do this? In a single grayscale image, a point may darken either because its albedo is low (a stain) or because its normal has tilted away from the light (a dent slope) — the two causes cannot be told apart within a single scalar. Photometric stereo uses multiple illumination directions to "stretch" this scalar into a vector equation, and magnitude and direction then part ways. This is exactly the root cause of single-illumination inspection repeatedly stumbling on "stains misjudged as dents."

This bears directly on the design of decision criteria for defect detection (Chapter 26): shape-type defects (dents, scratches, edge collapse) should be thresholded on the normal or on curvature derived from the normal, while appearance-type defects (stains, oxidation, print smudges) should be thresholded on the albedo. The two kinds of defect require different criteria, and photometric stereo conveniently provides the two raw images that allow them to be judged separately.

34.3 Height Integration

With a normal field in hand, one naturally wants to integrate it back into height. The surface slopes corresponding to the normal $\mathbf{n} = (n_x, n_y, n_z)$ are $p = h_x = -n_x/n_z$ and $q = h_y = -n_y/n_z$; solving the Poisson equation $h_{xx} + h_{yy} = p_x + q_y$ then recovers the height (this is of the same origin as the frequency-domain integration of Frankot–Chellappa in Chapter 11, while this chapter solves it in the spatial domain by SOR iteration). The integrated

height map is shown in Figure 34.5, and the experimental results are intriguing:

- **Slowly varying features are recovered well.** The 0.3 mm Gaussian dent integrates to −0.317 mm, less than 6% off the true value of −0.300 mm — smooth, shallow defects are exactly integration's strong suit.
- **Steep walls are severely underestimated.** The raised logo has a true height of 2.00 mm, yet integration yields only **1.16 mm**. The reason is honest and profound: the logo's side walls, sloping at about 76°, undergo **self-shadowing** under a 45°-slant light, with the walls either facing away from the light or covered by their own shadow, so their normals are "flattened" by the solver and the step height decays accordingly. This is not a bug but a genuine teaching point of photometric stereo under steep geometry — it excels at microscopic slope but is poor at restoring near-vertical steps.
- **Low-frequency drift.** The four corners of the flat bottom should be at equal height, yet after integration the height spans 0.094 mm. This is the intrinsic low-frequency drift of Poisson integration under Neumann boundaries: integration constrains only slope (the derivative of height), and a global slow tilt leaves almost no trace in the derivative, so it cannot be pinned down.

34.4 Lambertian Violations and Robustness

The Lambertian model is the bedrock of the whole solution, yet real surfaces are often non-Lambertian: metal, glaze, and oil films produce **specular highlights**, and protrusions cast **shadows**. A highlight is a bright spot far exceeding diffuse reflection, while a shadow is a dark region that should have had brightness but is zero — neither satisfies $I = \rho(\mathbf{n} \cdot \mathbf{l})$, and once one slips into the least-squares fit, it drags the entire solution of that pixel off course.

We add one more specular highlight region to the relief board and capture a 5th image containing the highlight, then

Remember the boundary of photometric stereo's height capability in one line: **it excels at microscopic relief, not at absolute height**. Slope information is its first-hand data; integration is only an after-the-fact reconstruction. The more high-frequency and slowly varying the relief (scratch depth, orange-peel texture), the more accurate it is; the more large-scale the absolute step height or overall warpage, the less reliable. For absolute height, go back to laser triangulation or deflectometry.

Figure 34.5: Height map obtained from the normal field by Poisson (SOR) integration. The slowly varying dent is recovered well, the logo's steep wall has its height underestimated due to self-shadowing, and a slight low-frequency drift exists at the four corners.

compare the normal error of three solution methods in the highlight region:

Configuration	Normal error in highlight region
4 lights (no highlight)	1.15°
5 lights (with highlight, naive least squares)	**11.81°**
5 lights (with highlight, robust rejection)	**1.15°**

Naive least squares treats that highlight measurement as trustworthy data, and the error explodes from 1° to nearly 12°. The approach of **robust photometric stereo** is extremely simple yet effective: first solve once using all measurements, find the term with the largest residual (most likely the highlight or shadow), reject it, and re-solve with the remaining measurements. In this way the error falls back to 1.15° — on par with the case of no highlight at all. The error maps in the middle (naive least squares) and on the right (robust rejection) of Figure 34.6 intuitively contrast the night-and-day difference between the two in the highlight region.

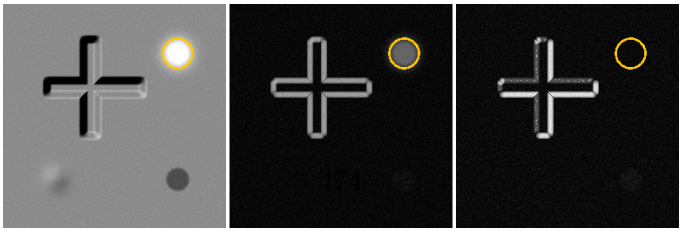


Figure 34.6: Lambertian-violation experiment: left is the 5th input containing the specular highlight, middle is the normal-error map of naive least squares (error explodes in the highlight region), right is the error map after robust rejection (the highlight region is

The recipe of robust photometric stereo can be condensed into one phrase: **more light sources + outlier rejection**. This is of the same origin as Huber robust line fitting in Chapter 14 — both acknowledge that "a few measurements are outliers" and use rejection or down-weighting to exclude them from the estimate. The only difference is that there the outliers came from defect edges, while here they come from highlights and shadows. The more light

produces **silent empty output** on this machine — it does not crash, returns code `rc=0`, but the `Nx/Ny/Nz/albedo` outputs are all 0×0 empty images, and it prints "Directory does not exist." to stderr (exactly the same failure signature as `PhaseMeasure` in Chapter 35, suspected to be a missing runtime resource directory). This chapter therefore records the defect faithfully with a subprocess probe, and switches the entire solution to a **hand-written per-pixel least squares** — which is exactly the mathematical substance of Section 34.1. The core snippet follows.

```
// Per-pixel solve g = albedo·n: min_g Σ_k (I_k - g·l_k)²
//   normal equation (Σ l_k l_k ) g = Σ I_k l_k
double M[9] = {0}, b[3] = {0};
for (int k = 0; k < K; ++k) {                  // K light sources
    double l[3] = { Lx[k], Ly[k], Lz[k] };     // known light directions
    for (int r = 0; r < 3; ++r) {
        b[r] += I[k] * l[r];                   // Σ I_k l_k
        for (int c = 0; c < 3; ++c)
            M[r*3 + c] += l[r] * l[c];          // Σ l_k l_k
    }
}
solve3(M, b, g);                               // solve 3×3 for g
// Robust: reject the term with the largest residual, then re-solve with the remaining  3 terms
if (robust && K >= 4) {
    int worst = argmax_k |I[k] - g·l_k|;
    re-accumulate M,b (skipping worst), then solve3 -> g;
}
double albedo = norm(g);                       // magnitude = albedo
n = g / albedo;                                // direction = normal (take n_z 0 facing the came
```

Height integration is implemented by per-point SOR iteration of the Poisson equation: compute the slopes p, q from the normals, take their divergence to obtain the right-hand side $f = p_x + q_y$, then iterate with relaxation factor $\omega = 1.9$ to solve $h_{xx} + h_{yy} = f$ (Neumann boundary). The complete runnable project is at `code/photometric_stereo/`.

Industry Case: Stamped Characters and Stains on Nameplates

A metal-nameplate inspection line had to judge two things at once: whether the stamped characters were clear and complete

(shape-type), and whether the surface was contaminated by oil stains or oxidation spots (appearance-type). The early single-illumination grayscale scheme made the shadow at the bottom of a stamp and a dark oil stain both appear as "dark regions" in the image, hard to tell apart — raising the threshold missed shallow stamps, lowering it misjudged oil stains as character defects, and the false-detection rate stayed high. Switching to 4-light photometric stereo resolved the problem at once: the normal map reflects only the stamping depth and edge steepness of the characters, while the albedo map reflects only the reflectance change of oil and oxidation. The two maps each set their own criteria, without mutual interference, and the false-detection rate dropped sharply. The lesson is clear: when "defects" include both shape-type and appearance-type, photometric stereo's ability to decouple the two into separate channels is irreplaceable — something no single-illumination scheme can deliver.

34.6 Summary

- **Photometric stereo measures the normal, not the height.** The camera stays still while light sources of known direction are switched; the Lambertian model $I = \rho(\mathbf{n} \cdot \mathbf{l})$ resolves the brightness variation into each point's orientation; 3 sources fix the solution, 4+ sources give least squares, and the magnitude of $\mathbf{g} = \rho\mathbf{n}$ is the albedo while its direction is the normal.
- **It can separate shape defects from appearance defects.** In this chapter's experiments the dent jumps to 8× the baseline in the normal channel while its albedo stays unchanged, and the stain pushes the albedo down to 55% while its normal does not move — a decoupling that single-illumination grayscale cannot achieve, and one that directly decides which channel the defect criterion should be built on.
- **Normals can be integrated back into height, but with limits.** Slowly varying shallow defects are recovered accurately (dent −0.317 vs −0.300 mm), steep walls are underestimated due to self-shadowing (logo 1.16 vs

2.00 mm), and low-frequency drift is an intrinsic product of the Neumann boundary — good at microscopic relief, poor at absolute height.

- **Robustness = more light sources + outlier rejection.** Highlights and shadows violate the Lambertian assumption, naive least squares explodes in error (11.81°), and after rejecting the measurement with the largest residual the error falls back to 1.15°, of the same origin as Huber robust fitting.
- **The SDK's photometric stereo interface is unusable on this machine** (silent empty output), and the hand-written per-pixel least squares is both the fallback and the very substance of this chapter's mathematics.

The foundational work of photometric stereo is Woodham's method of determining surface orientation from multiple illuminations (Woodham 1980); to address highlights and shadows that break the Lambertian assumption, Wu et al. model the problem as low-rank matrix completion and recovery, giving a convex-optimization solution for robust photometric stereo (Wu et al. 2011) that shares its spirit with this chapter's outlier rejection. A systematic treatment of photometric stereo, shape reconstruction, and robust estimation in industrial vision can be further consulted in the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

35 Phase Measuring Deflectometry

Put a piece of polished phone cover glass, a mirror-finished stainless-steel panel, or a plane mirror onto the production line, and the 3D imaging techniques built up in the preceding chapters fail en masse: the fringes projected by structured light (Chapter 32) no longer diffuse back to the camera but are reflected straight off to somewhere else by the specular surface; laser triangulation (Chapter 30) cannot find a stable scattered light stripe; binocular stereo has nothing to match either — what you see on a mirror is not surface texture at all, but a reflection of the surroundings. Light does not diffuse on a mirror, so every method that relies on "seeing the surface lit up" loses its footing.

Yet a mirror can do precisely one thing other surfaces cannot: serve as a **mirror**. If we let the specular surface reflect a screen displaying sinusoidal fringes, what the camera sees "in the mirror" is fringes distorted by the surface relief. Wherever the surface tilts a little, the fringes reflected into the camera shift a little there — the amount of fringe distortion directly encodes the surface **slope**. This is **phase measuring deflectometry (PMD)**: instead of illuminating the surface to measure height, it treats the surface as a reflecting mirror and infers slope from the phase shift of the fringes. Figure 35.1 is the raw image obtained from the surface under test reflecting the screen fringes, where the shallow bump and the scratch-like groove have already left visible local distortions on the regular fringes.

PMD and structured light (Chapter 32) are a **dual** pair: structured light requires the surface to be **diffuse** and measures **height** directly; PMD requires the surface to be **specular** and measures **slope** directly. The two use the very same phase-shifting and unwrapping machinery, yet serve two geometrically opposite classes of surface. Almost every property in this chapter has its mirror image in the structured-light chapter.

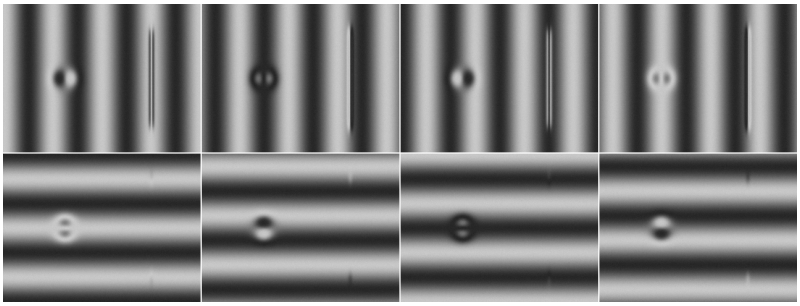


Figure 35.1: Raw phase-shifted images obtained from the specular surface under test reflecting the screen sinusoidal fringes. The top row is the 4-step phase shift of x-direction fringes, the bottom row the 4-step phase shift of y-direction fringes (screen distance $L = 200$ mm, fringe period $P = 16$ mm, sensor noise $\sigma = 2$). The orderly fringes are locally tugged at the bump (the circular spot on the left) and the scratch-like groove (the vertical line on the right) — these distortions are the fingerprint of surface slope.

35.1 Specular Reflection and Slope Encoding

The geometry of PMD is made up of three players: the **screen** that emits the fringes, the specular **surface** under test that reflects the fringes into the camera, and the imaging camera. The screen displays a spatial sinusoidal pattern at distance L from the surface; each line of sight from the camera hits a point on the surface, folds back to the screen according to the law of reflection, and lands at some screen coordinate — the screen phase at that coordinate is the phase observed by that pixel.

The key is how a surface tilt changes this light path. Let the local slope of the surface at some point be $s = \partial h/\partial x$, i.e. the surface is tilted by a small angle $\alpha \approx s$ relative to the reference plane. The **law of reflection** tells us: if the surface normal rotates by α, the reflected ray rotates by 2α — this is the **factor-of-2 deflection law** of deflectometry. The lateral landing point of the ray at screen distance L is therefore

shifted by about $2L\tan\alpha \approx 2L\,s$. So what the camera observes at that pixel is no longer the phase of the screen point straight below, but the phase of the point shifted by $2L\,s$:

$$\varphi_{\text{obs}}(x) \;\approx\; \varphi_{\text{screen}}(x + 2L\,\tfrac{\partial h}{\partial x}).$$

Subtracting from it the uniform screen phase $\varphi_{\text{screen}}(x) = 2\pi x/P$, the offset of the observed phase relative to the plane-mirror reference is proportional to slope: $\Delta\varphi = \varphi_{\text{obs}} - \varphi_{\text{ref}} = (2\pi/P)\cdot 2L\,s$. In other words,

$$s \;=\; \frac{\partial h}{\partial x} \;=\; \frac{P}{2\pi\cdot 2L}\,\Delta\varphi.$$

The "factor of 2" comes from the law of reflection: turn the mirror by α, the ray turns by 2α. This is also why PMD is so exquisitely sensitive to slope — a milliradian tilt of the surface is magnified at $L = 200$ mm away into a fringe displacement of $2L\alpha = 0.4$ mm, equal to 2.5% of the fringe period $P = 16$ mm, clearly distinguishable in phase. The geometry carries a first stage of amplification built in.

This equation is the root of all of PMD's properties: **it measures slope $\partial h/\partial x$, not height h.** Height can only be obtained indirectly by integrating the slope field (Section 35.3). This stands in sharp contrast to structured light — the phase of structured light is directly proportional to height, while the phase of PMD is proportional to the derivative of height. The derivative is sensitive to high-frequency relief and sluggish about absolute magnitude, and this trade-off runs through the whole chapter: it makes PMD astonishingly sensitive when detecting tiny relief (Section 35.4), yet also makes the absolute height depend on a drift-prone integration.

35.2 Reusing Phase Measurement

Since the measured quantity is the phase of the screen fringes, the machinery for recovering phase can be borrowed wholesale from structured light (Section 32.1). The screen displays 4-step phase-shifted fringes along x, the camera captures 4 frames $I_0 \dots I_3$, and the wrapped phase is computed by hand with the four-step phase-shift formula:

$$\varphi_{\text{wrap}} = \operatorname{atan2}(I_3 - I_1,\ I_0 - I_2),$$

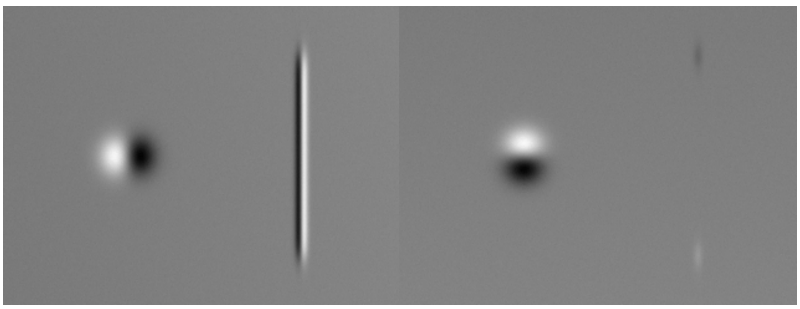
The modulation $B = \frac{1}{2}\sqrt{(I_3 - I_1)^2 + (I_0 - I_2)^2}$ measures the fringe contrast along the way, to be used later in Section 35.5. Because the deflection phase field is smooth (carrier about 0.08 rad/px, and the deflection introduced by a defect is well under π), single-frequency **spatial unwrapping** on a flat mirror is enough — no need for the dual-frequency hierarchical method of structured light (Section 32.2). One pass each for x and y yields the unwrapped phase in both directions.

As in the structured-light chapter, SciVision's `SciSvPhaseMeasure::DecodePatterns` (four-step phase-shift decoding) is **inert** in this SDK build — it returns code 0 but outputs an empty image. PMD is exactly its home turf, so the example still probes it once more with genuine bidirectional X+Y fringes and faithfully records its failure, after which the entire phase pipeline is written by hand (wrapped phase via `atan2` + spatial unwrapping).

The final step of measurement is to **subtract the plane-mirror reference phase**: render a separate set of phase-shifted images of an ideal plane mirror, solve for its phase φ_{ref}, and subtract it from the measured phase to cancel the carrier and the system geometry; the remaining $\Delta\varphi$ converts directly into the slope fields s_x, s_y. Figure 35.2 is the slope map in the two directions. The most intriguing feature is the **bipolar signature** of the bump in the x-direction slope map (Figure 35.2a): a 50 m Gaussian bump appears on the slope map not as a single bright blob, but as two lobes, one positive and one negative — because slope is the derivative of height, the up-slope of the bump is positive and the down-slope negative, with the slope crossing zero at the peak. In the measured neighborhood of the bump the x-direction slope swings between −0.0146 and +0.0141 rad, agreeing with the theoretical peak slope ±0.0143 rad of the Gaussian bump. The scratch-like groove on the right, extending along y and having slope only in x, shows up as a clean vertical line.

35.3 Slope Integration and Height

To obtain height, one must integrate the slope field back up — this is isomorphic to recovering height from the normal field in photometric stereo (Chapter 34): both hold the surface gradient and both need to recover a scalar height field. The most naive approach is to accumulate a trapezoidal integral



(a) x-direction slope s_x (b) y-direction slope s_y

Figure 35.2: The two-direction slope fields converted from the deflection phase minus the reference phase (display range ± 15 mrad). (a) x-direction slope: the 50 m bump appears as one positive and one negative bipolar lobe (slope is the derivative of height — up-slope positive, down-slope negative, zero-crossing at the peak), the scratch groove as a vertical line; (b) y-direction slope: the bipolar lobes of the bump rotate to the up-down direction, while the groove extending along y has almost no slope in y and nearly vanishes.

along a path: integrate s_y down the first column, then integrate s_x row by row to get the height along one path; switch to another path (first s_x along the first row, then s_y column by column) and compute again, averaging the two paths to suppress noise. With a smooth surface and small noise, this simple integrator does the job without the complexity of a Poisson solver.

Figure 35.3 is the height map obtained by integration. Quantitatively, the example measures the bump height as **46.83 m** using the central disk minus an outer-ring baseline (GT 46.71, nominal 50), and the groove depth as **10.13 m** using the groove bottom minus a side baseline (GT 10.16, nominal 10) — both close to the true values, with the residual deviation coming mainly from the finite-aperture baseline selection rather than measurement noise. The height RMS error in the flat region is only **0.228 m**: slope integration spatially averages the per-pixel noise, compressing the 1.6×10^{-4} rad slope noise down to sub-micron height noise.

Integration has an inherent weak spot: **low-frequency drift**. Slope noise accumulates along the integration path and piles up into a slowly undulating "terrain" on the height map; together with the integration constant itself being undetermined, PMD's absolute height and large-scale form are not reliable. This is exactly where it is complementary to structured light — structured light has accurate absolute height but poor high-frequency sensitivity; PMD is the reverse.

35.4 Sensitivity: Why Measuring Slope Is So Accurate

This is the heart of the chapter. PMD measures slope, and slope is an **angle** — angular quantities are extremely easy to measure precisely, because the screen distance L magnifies a tiny tilt into a sizable fringe displacement. Working through the noise derivation makes this clear. The phase noise of four-step phase shifting is $\sigma_\varphi = \sigma_N/(\sqrt{2}\,B)$; subtracting the reference from the measurement amplifies it by another $\sqrt{2}$, giving $\sigma_{\Delta\varphi} = \sigma_N/B$, so the slope noise is

$$\sigma_{\text{slope}} \;=\; \frac{\sigma_N}{B} \cdot \frac{P}{2\pi \cdot 2L}.$$

Substituting $\sigma_N = 2$, $B = 80$, $P = 16$ mm, $L = 200$ mm, the theoretical value is $\sigma_{\text{slope}} = 1.592 \times 10^{-4}$ rad, consistent with the measured 1.643×10^{-4} rad. Just how small is this angular noise? For a defect of base width w and height h, the

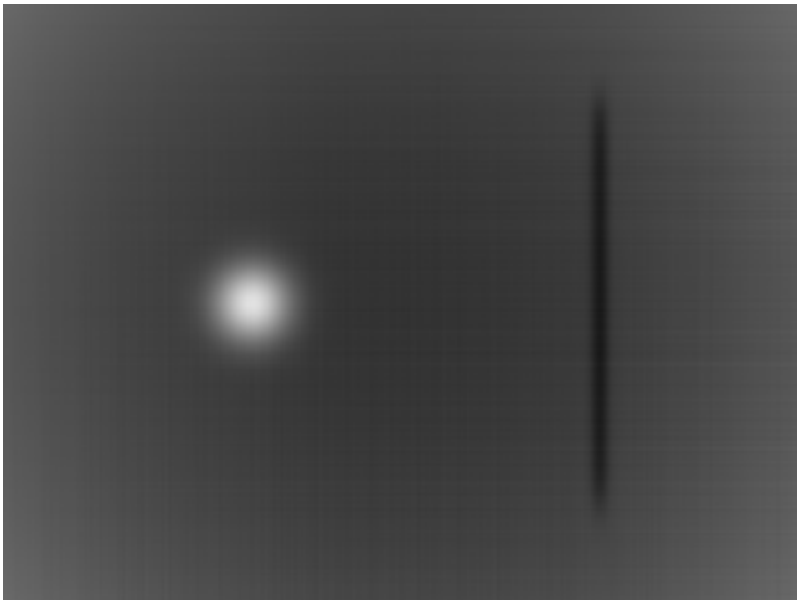


Figure 35.3: The height map obtained by two-path integration of the slope field (display range -15 to $+60$ m). The bump is reconstructed as 46.83 m (GT 46.71), the groove as 10.13 m (GT 10.16), with a flat-region height RMS error of 0.228 m.

maximum slope is about $4h/w$; the condition for it to be detectable is that this slope exceed the noise, i.e. $4h_{\min}/w \approx \sigma_{\text{slope}}$, giving the **minimum detectable height**

$$h_{\min} \approx \frac{\sigma_{\text{slope}}\, w}{4}.$$

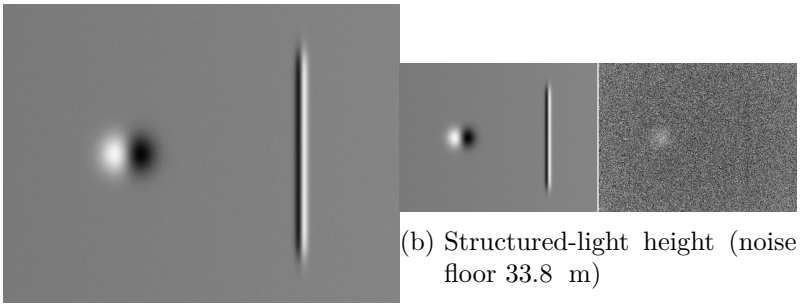
Substituting the measured σ_{slope}: a $w = 1$ mm defect gives $h_{\min} = 0.041$ m, $w = 5$ mm gives **0.205 m**, and $w = 20$ mm gives 0.821 m. Narrow, shallow microscopic defects can be measured down to **tens of nanometers** — far beyond the reach of methods that measure height directly.

The comparison is right there in Figure 35.4. The same surface and the same 50 m bump: the left image is the PMD slope map, the right is what structured light sees at its height noise floor (measured at **33.8 m** in ch32). The structured-light height noise floor is almost on the same order as the 50 m bump, so the bump is drowned in granular noise and barely discernible; on the PMD slope map, by contrast, the bump and the groove are strikingly clear. Quantitatively: for the same bump, the **PMD slope SNR is 87:1, while the structured-light height SNR is only 1.5:1**, a difference of about **59×**.

35.5 The Boundary of Specularity

The entire premise of PMD is specular reflection. Once the surface turns rough, the microfacets scatter the incident light, and the contrast of the specularly reflected fringes collapses accordingly. The example sweeps roughness through the diffuse fraction ρ, convolving the screen sinusoidal pattern with a Gaussian scattering lobe, so that the effective contrast decays as $B_{\text{eff}} = (1-\rho)B\exp[-\frac{1}{2}((2\pi/P)\cdot 2L\,\sigma_\theta)^2]$, and the phase noise $\propto 1/B$ explodes along with it:

There is no free sensitivity. PMD is exquisitely sensitive to slope (high-frequency relief), at the cost of an absolute height that depends on a drift-prone integration (Section 35.3) and is, conversely, sluggish about slow large-scale undulations. Structured light is exactly the opposite. In engineering the two are often **complementary**: structured light fixes the large-scale form, PMD fills in the microscopic high-frequency defects.



(a) PMD slope map

(b) Structured-light height (noise floor 33.8 m)

Figure 35.4: The same surface: PMD slope (left) renders the 50 m bump and the scratch clearly (SNR 87:1); structured light measuring height directly (right half, with a 33.8 m height noise floor superimposed) drowns the bump in granular noise (SNR 1.5:1), a sensitivity gap of about 59×. PMD measures slope, and therefore amplifies high-frequency relief.

ρ	Contrast B	Phase RMS (rad)	Slope RMS (rad)	$h_{\min}$ (w=5mm, m)
0.0	80.0	0.0255	1.626×10^{-4}	0.203
0.3	47.7	0.0364	2.314×10^{-4}	0.289
0.7	10.1	0.1449	9.222×10^{-4}	1.153

From $\rho = 0$ to 0.7, the contrast collapses from 80 to 10, the phase RMS rises by nearly 6×, and the minimum detectable height degrades from 0.20 m to 1.15 m. The three fringe images of Figure 35.5 make this collapse vividly clear: in the left image the fringes are crisp and the defect is distinct, while by the right image the fringes have turned into a hazy gray and the signal is all but buried. This is precisely the **mirror dual of the modulation story (Section 32.4)** of the structured-light chapter — structured light requires the surface to be diffuse enough and fails in specular regions due

to specular-highlight saturation; PMD requires the surface to be specular enough and its contrast collapses as soon as the diffuse component grows. A semi-specular surface is a dilemma for both: too bright and the mirror does not move, too rough and the diffuse signal is insensitive, and surfaces landing in this middle ground tend to give both methods trouble.

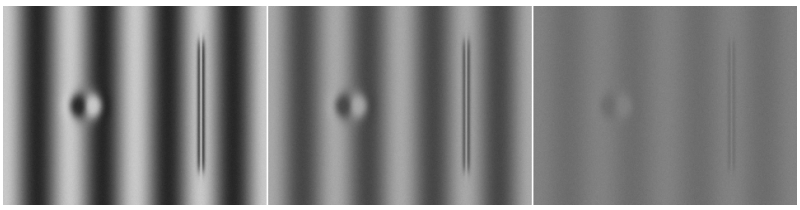


Figure 35.5: Fringe-contrast collapse under a roughness sweep (diffuse fraction $\rho = 0$ / 0.3 / 0.7, left to right). The rougher the surface, the lower the contrast of the specularly reflected fringes and the larger the phase noise, gradually burying the defect signal in noise — the dual of the structured-light modulation story.

35.6 SciVision Implementation

The entire PMD pipeline reuses the phase machinery of ch32. As covenanted, `SciSvPhaseMeasure::DecodePatterns` is still **inert** in PMD, its very home turf (returns 0, outputs an empty image), so the example probes it once more with a genuine bidirectional X+Y fringe set, faithfully records its failure, and then writes everything by hand. The key slope-extraction and integration snippet follows.

```
// 1. Four-step phase-shift wrapped phase (one set each for x/y),
//    subtract the plane-mirror reference phase -> slope field
double s = (double)im[3][i] - im[1][i];   // 2B*sin(phi)
double c = (double)im[0][i] - im[2][i];   // 2B*cos(phi)
phi[i]   = std::atan2(s, c);              // wrapped phase
// after unwrapping: dphi = phi_obs - phi_ref, convert to slope (rad)
//   via the factor-of-2 deflection law
```

```
slope[i] = (uObs[i] - uRef[i]) * P_SCR / (TWO_PI * 2.0 * L_SCR);

// 2. Two-path trapezoidal integration of slope -> height
//    (same form as photometric-stereo normal integration)
h1[i] = h1[i-1] + 0.5 * (sx[i] + sx[i-1]) * PX;   // path 1: integrate sx row by row
h2[i] = h2[i-W] + 0.5 * (sy[i] + sy[i-W]) * PX;   // path 2: integrate sy column by column
h[i]  = 0.5 * (h1[i] + h2[i]);                    // average the two paths to suppress noise
```

The `P_SCR/(TWO_PI*2.0*L_SCR)` term in the slope conversion is exactly $P/(2\pi \cdot 2L)$, translating the deflection phase difference into slope; the integration step `PX` is the physical pixel scale. One calibration point: the geometric accuracy of PMD hinges on the **relative poses of screen, camera, and reference plane** — the screen-to-surface distance L, the physical scale of the screen pixels, and the camera intrinsics all need to be calibrated beforehand (echoing Chapter 5), and any error in any of them propagates linearly into the slope through the deflection law. The complete project that generates all the images in this chapter is located at `code/deflectometry/`.

Industry Case: Micro-Defects on Phone Cover Glass

What needs to be inspected on polished phone cover glass is sub-micron microscopic relief: orange peel, polishing marks, shallow scratches, whose height undulations are often only a few tenths of a micron. Measuring height directly with structured light, the height noise floor on the order of 33 m buries these defects completely, leaving nothing to discriminate; switching to PMD to measure slope, the geometric factor-of-2 deflection law plus the magnification of the screen distance amplifies the sub-micron high-frequency relief into a clearly discernible slope signal, and the defects become obvious at a glance. But the cover glass is not specular everywhere: the chamfered edges and the silk-screened ink regions are diffuse or semi-specular, where the fringe contrast collapses and the phase noise explodes — they must be masked out separately and processed apart, not lumped together with the specular regions. The lesson is direct: for microscopic-relief inspection of specular or mirror-like surfaces, PMD's slope sensitivity is something

structured light cannot give; but PMD's Achilles' heel is also its specularity — non-specular regions must first be identified, then isolated.

35.7 Summary

- **PMD uses the mirror as a mirror**: it lets the specular surface under test reflect the screen fringes and infers surface slope from the fringe phase shift — the remedy precisely for the mirror objects on which structured light, laser, and binocular stereo all fail, and the geometric dual of structured light.
- **The factor-of-2 deflection law is the root of everything**: a surface tilt α deflects the reflected ray by 2α, $\varphi_{\text{obs}} \approx \varphi_{\text{screen}}(x + 2L\,\partial h/\partial x)$. PMD measures slope rather than height; height requires integrating the slope field (isomorphic to photometric stereo), at the cost of a drift-prone absolute height.
- **Measuring slope makes it exquisitely sensitive**: $h_{\min} \approx \sigma_{\text{slope}} w/4$, only 0.205 m at $w = 5$ mm. For the same 50 m bump, the PMD slope SNR of 87:1 versus the structured-light height SNR of 1.5:1 is about a 59× gain — PMD amplifies high-frequency relief.
- **Specularity is a hard boundary**: as the surface roughens, the fringe contrast collapses and the phase noise explodes ($h_{\min}$ degrades 5× at $\rho = 0.7$), the mirror image of the structured-light modulation story; a semi-specular surface is troublesome for both methods.
- **Reuse the phase machinery, mind the calibration**: phase shifting and unwrapping follow ch32 (`DecodePatterns` inert, written by hand), and the accuracy hinges on the screen-camera-reference-plane geometric calibration.

The pioneering work on measuring specular free-form surfaces with phase measuring deflectometry is the paper by Knauer, Kaminski, and Häusler (Knauer, Kaminski, and Häusler 2004); the principles, calibration, and application advances of the field are comprehensively surveyed by Huang et al. (Huang et al. 2018). The engineering treatment of

deflectometry and 3D measurement of specular surfaces can be further consulted in the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018).

36 Confocal Imaging and Focus Variation

By the time Part VIII reaches this point, the previous five 3D-imaging techniques have each leaned on one geometric or optical constraint to convert height into a measurable image quantity: stereo relies on disparity, structured light and deflectometry rely on phase, laser triangulation on displacement, photometric stereo on shading. They share one thing in common — all of them treat the lens's **depth of field** as an ever-welcome ally, the bigger the better, wishing the entire measurement range could be sharp everywhere. This chapter does the opposite, turning the "defect" of limited depth of field into the very means of measurement: the lens renders only the object plane that is exactly in focus sharp, so **scan layer by layer along the optical axis, see at which layer each pixel is sharpest, and the Z position of that layer is the height of the point**. In a sentence — "wherever it is sharpest, that is where the object sits in height." This is **focus variation (also called shape from focus)**, together with its cousin, **confocal microscopy**, which physically rejects out-of-focus light with a pinhole. They are slow and have a small field of view, yet they are the workhorses of microscale topography measurement: surface roughness, the cutting-edge radius of a tool, the step heights of microstructures — measuring these down to the submicron relies on this family of methods.

Figure 36.1 is this chapter's starting point — slices of the same staircase surface captured at three focus heights. Note carefully: as the focus plane rises, the left, middle, and right step segments sharpen in turn, while the patch in the figure that stays smoothly blurred throughout is exactly the "failure region" that the second half of this chapter will discuss at length.

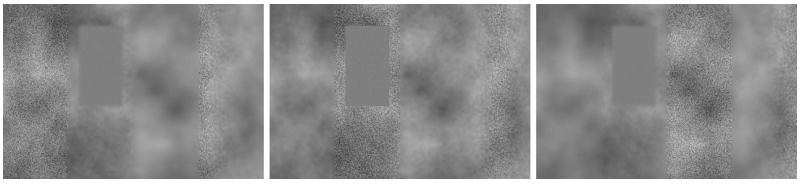


Figure 36.1: Three slices from the focus stack (z = 3 / 53 / 103 m). The staircase surface has three plateaus (heights about 0 / 50 / 100 m) plus a ramp on the right; when the focus plane is below, step A (left) is sharpest, when in the middle step B is sharp, and when raised step C (right) is sharp — the sharpness of a given pixel rises and then falls with z. The gray rectangle inside plateau B is a texture-poor region, smooth in all three slices, foreshadowing where focus variation will fail.

36.1 Principle of Focus Variation

Recall the discussion of depth of field and sharpness in Chapter 3: an object point converges to a sharp point on the image plane only when it lies within the depth-of-field range near the focus plane; the farther it deviates from the focus plane, the larger the circle of confusion, and the blurrier the image. To quantify this: if the object point's height is h and the focus plane is at z, then the larger the defocus $|z - h|$, the more the high-frequency detail in the point's neighborhood is wiped flat. This chapter's synthetic data is modeled exactly on this physics — each slice applies to every pixel a space-varying Gaussian blur of standard deviation $\sigma = 0.4\,\text{px}$ per 10 m of defocus, so that at focus $\sigma \to 0$ and the texture is sharpest.

The measurement procedure then becomes clear: acquire a stack of images at fixed intervals along the optical axis, forming a **focus stack**; for each pixel, compute a **focus measure** value layer by layer, yielding a sharpness curve along z; the z corresponding to the curve's **peak position** is the height of that pixel.

The sharpness operator has to answer "how sharp is this

neighborhood," which in essence is measuring local high-frequency energy — sharing the same root as the gradient operators of Chapter 13. Two families are common in engineering: **Tenengrad** (the sum of squared Sobel gradient magnitudes) and the **sum of modified Laplacian (SML)**. This chapter's implementation adopts the latter: first, for each pixel, take the sum of the absolute values of the second differences in the x and y directions,

$$\mathrm{ML}(x,y) = \big|\, 2I(x,y) - I(x-1,y) - I(x+1,y) \,\big| + \big|\, 2I(x,y) - I(x,y-1) - I(x,y+1) \,\big|,$$

the reason for taking absolute values before summing (rather than summing directly as the ordinary Laplacian does) is to avoid the x- and y-direction curvatures cancelling each other when they have opposite signs; then sum ML within a 9×9 window around each pixel to obtain that pixel's sharpness at that layer. The window sum is the crucial step — the single-pixel second difference is extremely sensitive to noise, so neighborhood aggregation is needed to "flatten and smooth" the sharpness curve, making the peak stable.

Figure 36.2 plots this sharpness curve. The blue one comes from a textured, sharp pixel (plateau C, true height 100 m): the sharpness rises then falls with z, forming a clear peak at $z \approx 100\,\mu$m — wherever the peak is, the height is. The gray one is left for Section 36.3.

Why does focus variation measure only height **relative** to the stack, with no external calibration scale needed? Because it does not ask "how far is this point from the camera" but "at which layer is this point sharpest" — the layer index times the known slice spacing is the physical height. The Z-axis accuracy is therefore determined entirely by the stepping accuracy of the scanning stage, independent of the lens's lateral calibration. This is also why it can easily reach submicron at the microscale: getting one mechanical step accurate is far easier than calibrating a whole imaging geometry accurately.

36.2 Quantization and Subpixel

Taking "the layer with maximum sharpness" directly as the height immediately runs into an accuracy ceiling: the slice spacing itself is the **quantization** step. This chapter's stack has 21 layers spaced 10 m apart, so the coarse height can only be …, 90, 100, 110… these integer multiples of 10 (offset by a -7 m stage-origin shift), and any detail with a height difference of less than 10 m gets merged into the same layer. Figure 36.3 plots the coarse height map — a ramp that should be a continuous transition is sliced into 10- m-wide contour

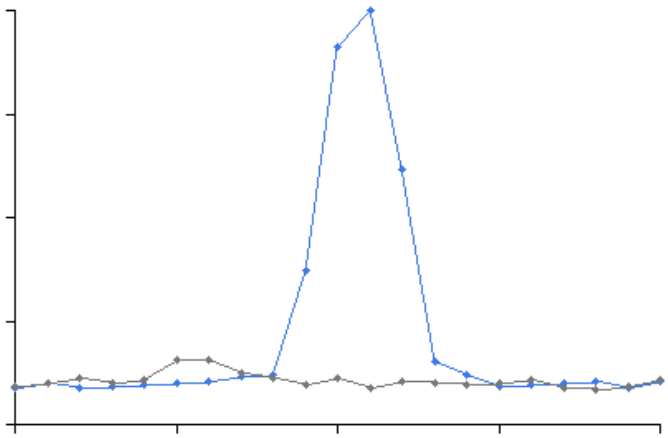


Figure 36.2: Sharpness curves for two pixels (horizontal axis is focus height z, vertical axis is normalized SML sharpness; both curves normalized to the same scale). Blue: a textured, sharp pixel, the curve forms a clear peak located at about $z = 100\,\mu$m, corresponding to its true height; gray: a pixel in a texture-poor region, the curve is low and flat with no peak, and its peak position is decided at random by noise — this is the core contrast between where focus variation holds and where it fails.

terraces (terracing), looking just like a topographic contour map. This is exactly the shape of quantization error.

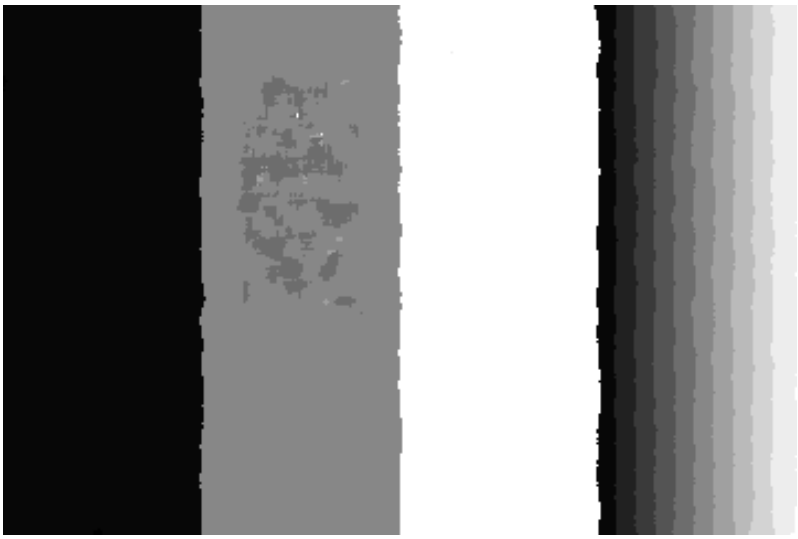


Figure 36.3: Coarse height map obtained by peak-layer quantization (grayscale-encoded height, 0–100 m). The three plateaus are clearly distinguished, but the ramp on the right that should be smooth is cut into staircase-like contour bands — each band's width corresponds to one 10 m slice spacing, the quantization trace left behind by "take the nearest layer."

The way to break past quantization is the **subpixel triad** idea that recurs throughout this book (used by the edges of Chapter 14 and the calipers of Chapter 20): the true peak almost never lands exactly on a sample point, but its true position is hidden in the shape of the curve near the peak. Concretely, fit a parabola to the sharpness values $(f_{k_p-1}, f_{k_p}, f_{k_p+1})$ of the peak layer k_p and its two neighbors, and take the vertex as the sub-layer peak position:

$$\delta = \frac{1}{2}\,\frac{f_{k_p-1} - f_{k_p+1}}{f_{k_p-1} - 2f_{k_p} + f_{k_p+1}}, \qquad h = (k_p + \delta)\cdot \Delta z + z_0,$$

where $\delta \in [-1, 1]$ is the sub-layer offset relative to the peak

layer and $\Delta z = 10\,\mu$m is the slice spacing. This step liberates the height from "integer multiples of 10 m" to a continuous value. Figure 36.4 is the refined height map — the terraces are gone, the ramp is restored to a smooth transition, and the three plateaus become clean.

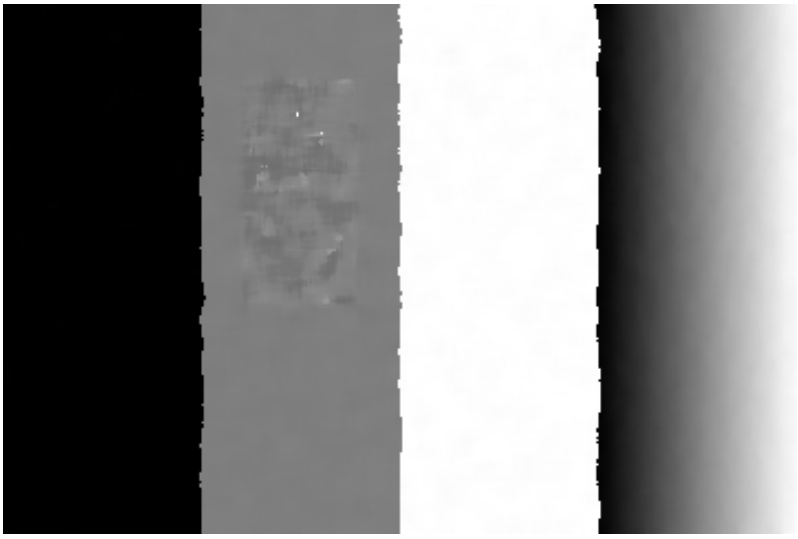


Figure 36.4: Height map after parabolic subpixel interpolation. The quantization terraces of Figure 36.3 are eliminated, the ramp is smooth and continuous, and the plateau noise drops markedly — from the same set of sharpness curves, sub-layer interpolation of the peak position alone raises the Z resolution by nearly an order of magnitude.

The numbers are honest. On the three flat plateaus, the height RMS error drops from the quantized **3.000 m to the subpixel 0.342 m, an improvement of 8.8×**; the ramp region on the right drops from 2.933 m to 0.442 m, an improvement of 6.6×. The step heights (differences of the mean subpixel heights of the three plateaus) measure B−A = **49.935 m**, C−B = 50.044 m, C−A = 99.980 m, essentially exact against the true values 50 / 50 / 100 m — submicron step measurement, "interpolated" out of a 10 m quantization grid just like this.

The premise of parabolic interpolation is that the curve near the peak is approximately symmetric and twice differentiable. When the slice sampling is too sparse (the peak spans only two or three layers) or the curve is gnawed ragged by noise, this assumption loosens and the δ estimate distorts accordingly. Engineering offers two countermeasures: take the slice spacing denser when scanning (let the peak span more than 5 layers),

36.3 Texture Dependence

Focus variation has one fatal premise, already spoiled by the gray curve of Figure 36.2: **the sharpness curve must have texture before it can have a peak**. The sharpness operator measures the decay of local high-frequency energy with defocus; but if a pixel's neighborhood has no high-frequency content of its own — a uniform, polished, textureless surface — then whether in focus or not, its sharpness stays close to a low, flat constant. With no peak in the curve, the whole logical chain of "peak position = height" breaks: the algorithm can only pick some maximum at random from a field of noise as the peak, and the height it gives is purely random.

Figure 36.5 is the confidence mask: using a peak prominence (normalized peak height) of 0.5 as the threshold, it paints the untrustworthy pixels black. The result is intuitive — the texture-poor patch inside plateau B is masked out as a whole. The quantitative contrast is equally stark: the valid rate of the texture-poor region is only **18.7% (1232/6600)**, whereas the region with normal granite procedural texture reaches a valid rate of **100% (14700/14700)**. This failure region is not a contrived counterexample; it is simply a textureless patch on the same solid surface — in reality, polished surfaces, mirror surfaces, and large smooth painted surfaces are full of it everywhere.

There are only two countermeasures, just as for stereo and structured light: either **actively create texture** — project structured illumination, spray a matting agent, "plant" high frequency onto the surface; or **honestly admit failure** — use a confidence threshold to reject the untrustworthy points, never letting a random peak position contaminate downstream measurement. Taking the garbage heights of a textureless region as real is far more dangerous than leaving a hole.

"No texture → failure" is the same hurdle running through the 3D part of this book, only wearing three different faces: the stereo of Chapter 31 cannot find left-right correspondences in textureless regions and the depth map opens holes; Chapter 32 simply actively "prints" striped texture onto the surface to bypass it; this chapter's focus variation gets no sharpness peak in textureless regions. The physical mechanisms of failure differ across the three (matching ambiguity / no passive texture / no high frequency to decay), but the root cause is the same: **the surface has no local structure for the algorithm to "grab."** Remember this isomorphism, and on encountering a smooth surface you will reflexively ask first, "where does my texture come from?"

Figure 36.5: Confidence mask (white = trustworthy, black = rejected). The vast majority of textured regions are judged trustworthy; the texture-poor block inside plateau B turns black in a continuous patch — there the sharpness curve is flat with no peak, the peak position is untrustworthy, and it is faithfully rejected by the prominence threshold.

36.4 Optical Sectioning in Confocal

The previous sections were all about focus variation, relying on an **algorithm** to find the sharpness peak in the focus stack after the fact. True **confocal** instead solves the same problem at the **optical physics** level, and the two must be honestly distinguished — this chapter's experiment does the former, while the latter described in this section has no accompanying simulation.

The core of confocal is a pair of **conjugate pinholes**. A point light source illuminates through a pinhole and is focused onto a point on the object plane; the light reflected or fluoresced from that point is then converged by the objective and can reach the detector only by passing through a **detection pinhole** that is optically conjugate to the illumination pinhole. The key is this: only light from the point on the **focus plane** converges exactly onto the detection pinhole and passes through smoothly; out-of-focus light from above or below the focus plane forms a diffuse spot at the pinhole plane, and the vast majority of it is blocked by the pinhole. Thus the signal the detector receives comes almost entirely from the thin, current layer in focus — this is **optical sectioning**. Scanning point by point (point-scanning confocal sweeps the whole field with a galvanometer or spinning disk) and then changing the focus-plane height layer by layer, one can "slice" out the 3D structure layer by layer.

Compared with focus variation, the difference is plain at a glance: **confocal physically rejects out-of-focus light in the light path with a pinhole**, so each layer's signal itself contains only focus-plane information, and the axial resolution is set by the optical diffraction limit, reaching submicron or even better; **focus variation relies on an algorithm to find the sharpness peak after the fact in the full image containing defocus**, so out-of-focus light always participates in imaging, and the accuracy is constrained by the sharpness operator and the texture quality. Each has its niche: confocal is high in accuracy with clean sectioning, but point-scanning is slow and the equipment expensive, and it still requires the measured surface to return enough signal; focus variation

works with just an ordinary microscope plus a Z-scanning stage, and since the area sensor images a whole layer at once it is far faster, at the price of a heavy dependence on surface texture. In a sentence — **choose confocal when accuracy comes first, choose focus variation when speed comes first and the surface has texture**.

36.5 SciVision Implementation

This is worth recording specially: after a string of SDKs in this book's 3D part that "fell silent with no output / crashed" (phase measurement, laser triangulation, and photometric stereo, not one of which was usable), `SCIMV::SciSv3DFocus` is the **rare usable case** — it really did return a valid depth map. The procedure is two steps: `CalGradArray` first computes the focus stack layer by layer into a stack of gradient (sharpness) maps, and `CalDepthMapByFocusStack` then outputs a `SciRangeImage` depth map from it.

```
SCIMV::SciSv3DFocus fo;
SciImageArray grad;
fo.CalGradArray(arr, 21, &grad);              // per-layer sharpness (gradient) maps of the fo
SciRangeImage depth; SciImage color;
fo.CalDepthMapByFocusStack(arr, grad, 31, 31, // smoothing window 31×31
                           1.0, 1.0, ZSTEP,   // x/y resolution, z slice spacing (m)
                           0, false, 1,
                           &depth, &color);
double z = depth.GetValue(row, col) * depth.ResolutionZ() + depth.OffsetZ();
```

But it has one clear shortcoming: **the depth is quantized to the nearest slice, with no subpixel**. Verifying on the sample points shows that the heights the SDK gives land exactly on the slice integer multiples 10 / 60 / 110, with none of the sub-layer interpolation of Section 36.2. This constitutes an honest and rare **favorable comparison**: the hand-written parabolic interpolation (submicron) is strictly better than the SDK's 10 m quantized output — not because the SDK crashed, but because it only got as far as "take the nearest layer." This chapter's code therefore walks on two legs: use

the SDK to verify that this stack-based height-measurement path works, then use a hand-written sharpness operator plus peak interpolation to push the accuracy to submicron. Below are the two core fragments of the hand-written part — sharpness (SML window sum) and the parabolic interpolation of the peak:

```
// modified Laplacian: take absolute values of the x/y second differences before summing (to av
float lx = std::fabs(2.0f*I[c] - I[c-1] - I[c+1]);
float ly = std::fabs(2.0f*I[c] - I[c-W] - I[c+W]);
ml[c] = lx + ly;                              // then sum within a 9×9 window to get the SML sh

// parabola vertex → sub-layer peak position    [-1,1]
double den = a - 2*b + c;                     // a,b,c = sharpness of the layers left-of-peak ,
if (std::fabs(den) > 1e-9) delta = 0.5*(a - c)/den;
double h = (kp + delta)*ZSTEP + ZBASE;        // continuous height ( m)
```

Industry Case: Micro-Measurement of Cutting-Tool Edges

The edge quality of a carbide milling cutter directly determines its cutting life, requiring micron-level measurement of the edge-arc radius and chipping notches. A production line used focus variation under a microscope to quickly build a focus stack of the edge, producing a height map in a few seconds, a cycle time far better than point-by-point confocal scanning. The problem arose on the polished back of the edge: there the surface was smooth, the local texture insufficient, the sharpness curve flat, and the height map showed patches of random jumps, “measuring” gullies that simply do not exist into what should be a smooth edge face. The countermeasure had two layers: first, add a beam of structured illumination to the edge, projecting onto the smooth face high-frequency texture for the sharpness operator to grab; second, use a peak-prominence threshold to reject the points that are still untrustworthy, preferring to leave holes over outputting garbage. The lesson is direct — **focus variation’s speed advantage is premised on texture; when you meet a textureless place, either create texture or switch to confocal**.

36.6 Summary

This chapter turned "limited depth of field" from a defect into a means of height measurement, with the key points as follows.

- **The peak is the height.** Acquire a focus stack along the optical axis, compute a per-layer sharpness curve for each pixel, and the Z corresponding to the peak position is the height; the sharpness operator (Tenengrad / modified Laplacian) is in essence a measure of local high-frequency energy, sharing the same root as the edge gradient.
- **The slice spacing is the quantization step.** Taking the peak layer directly can only give staircase-like heights at integer multiples of 10 m; a parabolic interpolation over the three layers near the peak yields the sub-layer peak position, and this chapter measured the flat-region RMS dropping from 3.000 m to 0.342 m (8.8×), with step measurements accurate to 49.935 / 50.044 / 99.980 m.
- **Focus variation depends heavily on texture.** The sharpness curve of a textureless region is flat with no peak, and the peak position is decided by noise (texture-poor region valid rate 18.7% vs textured region 100%); it must be rejected with a confidence threshold, or remedied by actively projecting texture — this is the same hurdle as the textureless failure of stereo and structured light.
- **Confocal and focus variation share a root but diverge in path.** Confocal relies on conjugate pinholes to optically reject out-of-focus light and do hard sectioning, submicron in accuracy but slow; focus variation relies on an algorithm to find the sharpness peak after the fact, fast but texture-dependent. Choose the former when accuracy comes first, the latter when speed comes first and there is texture.
- **A rare usable SDK case.** `SciSv3DFocus` returns a valid depth map, but quantized to the nearest slice with no subpixel — hand-written peak interpolation is strictly better, constituting an honest favorable comparison.

The foundational work of focus methods is Shape from Focus proposed by Nayar and Nakagawa (Nayar and Nakagawa

1994); a systematic comparison and evaluation of the various sharpness (focus-measure) operators is given by Pertuz et al. (Pertuz, Puig, and Garcia 2013). A systematic treatment of focus variation and confocal (the comparison of sharpness operators, the details of confocal optics) can be further consulted in the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018). With this, all six 3D-imaging techniques of Part VIII are complete: stereo and focus variation are passive / weakly active, while structured light, laser triangulation, photometric stereo, and deflectometry are active; laser triangulation and structured light target the macroscopic millimeter scale, while confocal specializes in the microscopic micron scale; the first five each serve diffuse surfaces, while deflectometry alone guards the specular. **No single one of the six techniques does it all — each has its own failure boundary (no texture, specular interreflection, occlusion, non-Lambertian, transparent, speed), and the essence of 3D selection is to first recognize which boundary your workpiece falls outside of.**

Part IX

3D Processing

This part covers the processing and application of 3D data: point cloud representation and preprocessing, ICP registration and 3D rectification, and the complete workflows of 3D matching, 3D measurement, and 3D inspection.

37 Point Cloud Fundamentals

Part VIII was about how to turn the real world into three-dimensional data: laser triangulation (Chapter 33), structured light, phase shift, focus stacking — each rests on its own physics, but in the end they all deliver the same kind of product, a heap of three-dimensional points carrying (x, y, z) coordinates. Chapter 30 strings this whole acquisition chain together. Starting with this chapter, Part IX, the topic switches to how to **process** the three-dimensional data already in hand: denoising, registration, measurement, fitting. And the object of all this processing, the overwhelming majority of the time, is the **point cloud**.

A point cloud is the "pixel" of the three-dimensional world, but it is far harder to handle than a two-dimensional image. A 2D image is a regular grid, where each pixel's neighbors — which ones, how many — are fixed directly by its row and column indices; a point cloud, however, is a bag of **unordered** points, where who is adjacent to whom is nowhere written into the data structure and must be computed for yourself. It is **sparse**: outside the surface, the space holds no record at all. It is **noisy**: every point jitters along the measurement direction. And it has **outliers**: arc light, glare, and multipath conjure up a batch of "**flyers**" hanging in mid-air out of nothing. This chapter lays out all four of these properties in a single synthetic scene (Figure 37.1, Figure 37.2) and uses it to explain three of the most fundamental things: which forms three-dimensional data takes, how to find nearest neighbors fast on an unordered point set, and how to clean up the flyers.

The whole scene comprises 20200 points: 11000 on the ground, 2600 on the cube top, 3600 on the four vertical sides, 2800 on the spherical cap, plus 200 outlier flyers. All surface points have Gaussian measurement noise of $\sigma = 0.10$ mm added along their respective normals, with a height range of

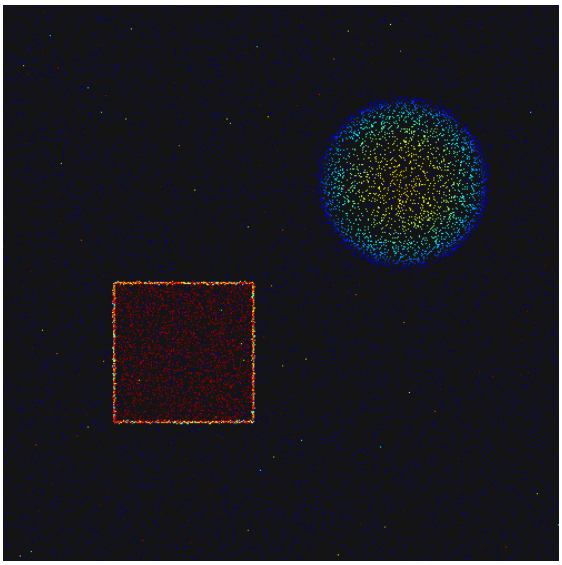


Figure 37.1: Top view of the synthetic point cloud (looking down the z axis), colored by height with a jet encoding: blue is the ground at $z \approx 0$, the red square at lower-left is the cube top at $z = 18$ mm, and the round patch at upper-right is the spherical cap, which forms a dome — low at the rim (blue) and high at the center (warm). The sparse stray dots scattered across the whole figure are the 200 floating flyers.

$z \in [-0.43,\ 43.92]$ mm (the surfaces themselves only reach 18 mm; everything higher is all flyers). This dataset is generated deterministically by the project in `code/point_cloud_basics/` using a fixed random seed, and every number later in this chapter comes from its actual run.

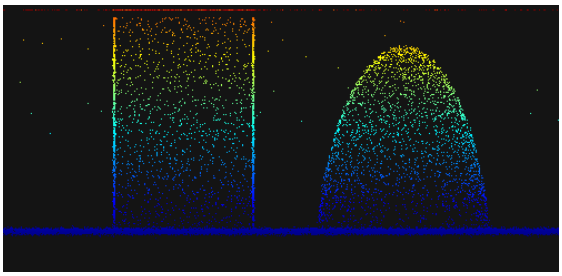


Figure 37.2: Side view of the same point cloud (XZ projection): the horizontal blue band at the bottom is the ground baseline, the vertical column from blue to red in the middle is the cube (the sidewall points link the blue ground to the red top), the spherical cap on the right forms a dome (low blue at the bottom rim, high warm at the top), and the scattered points above are the high-hanging flyers.

37.1 Forms of Three-Dimensional Data

One and the same three-dimensional object can be recorded with three fundamentally different data structures, and in engineering you must keep them straight.

The first is the **point cloud**, the unordered point set $\{\mathbf{p}_i = (x_i, y_i, z_i)\}$ mentioned above, optionally with a color or normal attached to each point. It is the most general form: it can represent surfaces of arbitrary topology, and multi-view stitching and free-form surfaces give it no trouble. The price is that it is unordered and irregular, so any "find the neighbors" operation must lean on an extra spatial index (the subject of the next section).

The second is the **range image**, also called a depth map or a 2.5D image. It is a regular two-dimensional grid, where each

grid cell (u, v) stores one height value z — essentially "an image that treats z as gray level." Its biggest advantage is regularity: neighbor relations are given directly by the grid indices, all the filtering and morphology operators of two-dimensional images (Chapter 6) can be carried over and used unchanged, and storage is compact.

The third is the **mesh**, which explicitly records the connectivity of the surface with vertices plus triangular faces. It mainly serves rendering, finite-element analysis, and collision detection; it sees relatively little use in industrial measurement, so this chapter does not develop it.

A point cloud and a range image can be converted into each other, but this conversion is **not lossless**. From range image to point cloud is simple: each valid grid cell is restored to a three-dimensional point according to its indices and height. The other way, generating a range image from a point cloud, first picks a projection direction (here we take the top view, looking along $-z$), bins the XY plane into a regular grid, and keeps only one representative z per bin. The problem lies precisely in this "keep only one."

The name "2.5D" pinpoints the awkward position of the range image: it has a complete x, y plane and a z, looking three-dimensional, yet it stipulates that each (x, y) can have only **one** z. A true 3D surface, at a vertical wall, an overhanging face, or a cavity, lets a single line of sight pass through multiple z values, and that is exactly what a range image cannot express — it can only record "the nearest (or highest) layer seen from some fixed viewpoint."

I binned these 20200 points onto a 200×200 top-view grid, taking the highest z among the points falling into each bin, with the result in Figure 37.3: the gridding artifacts are plainly visible, the ground has been quantized into a mosaic, the spherical cap is still there, and the cube top is there too — but the cube's **vertical sidewalls have vanished entirely**, and so has the ground directly beneath the cube and beneath the spherical cap, the part that was occluded. Statistically, of the original 20200 points, after reprojection only **12585 bins survive, losing 7615 points (37.7%)**. These lost points are not noise; they are real samples of a real surface, and they disappear purely because they share the same (x, y) with other points whose z is higher — the sidewall points are covered by the top points, and the occluded ground points are covered by the object above.

This 37.7% is not a number picked at random; it is a lesson on the "essence of a single viewpoint." In Chapter 33, laser triangulation directly produces a range image, and at that point it naturally sees only the surface that can be

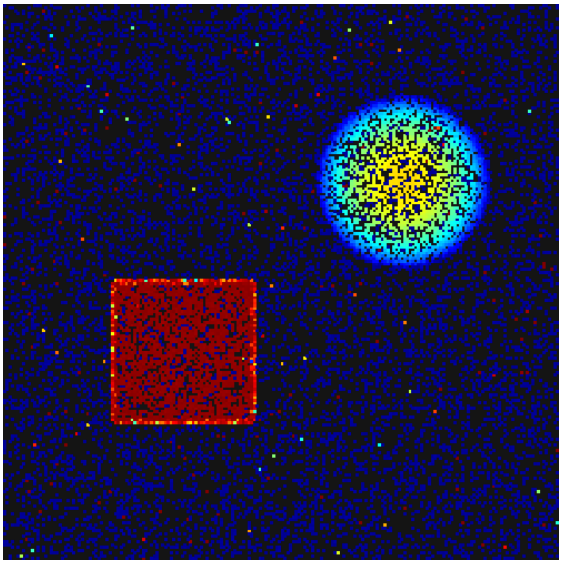


Figure 37.3: A 2.5D range image reprojected by binning the point cloud onto a 200×200 top-view grid and taking the highest z per bin. The grid quantization is plainly visible; the cube top (red) and the spherical cap are still there, but the vertical sidewalls and the occluded ground beneath have vanished as whole swaths — this is the information loss of 2.5D relative to true 3D.

illuminated from the laser's viewpoint — the back, the sidewalls, and the cavities were in shadow from the start. The 2.5D limitation of the range image and the single-viewpoint limitation of the acquisition method are two ways of saying the same thing. To obtain a complete 3D you must either scan from multiple viewpoints and then register (Chapter 39), or simply work with point clouds the whole way and never collapse to a range image. So a practical engineering rule is: **if you can use a point cloud, do not convert to a range image prematurely**; only when you are sure that a single viewpoint suffices, and you want to borrow the mature operators of two-dimensional images, is the regularity of the range image worth trading away that 37.7% of information for.

In SciVision these two forms correspond, respectively, to `SciPointCloud` (points + optional color + optional normal, with PLY/PCD/OBJ read/write) and `SciRangeImage` (`ushort` height data plus `resolutionX/Y/Z` and `offsetZ`, stored as a 16-bit PNG). After this chapter's scene is loaded into a `SciPointCloud`, `Length=20200`, and the saved `cloud_raw.ply` carries per-point height color and can be opened directly in any point-cloud viewer.

37.2 Spatial Indexing: KdTree

A point cloud is unordered, and the first, most fundamental engineering problem this brings is: given a query point, how do you find its nearest several neighbors? This "**nearest neighbor** query" is the bedrock of nearly all three-dimensional algorithms — registration relies on it to build point pairs (Chapter 39), filtering relies on it to define neighborhoods, and normal estimation, feature description, and surface reconstruction are no exceptions. The most naive approach is brute-force scanning: for each query point, compute its distance to all N points and then sort, $O(N)$ per query, $O(MN)$ for M of them. With N in the tens of thousands and M in the tens of thousands, this is tens of billions of distance computations — unacceptably slow.

The **kd-tree** brings this down to logarithmic complexity. Its

idea is to recursively bisect space with hyperplanes perpendicular to the coordinate axes: at the root node, take the median along the x coordinate, send the smaller to the left subtree and the larger to the right; the next level switches to the y axis, the level below to the z axis, and so the cuts proceed with the axis rotating, until each leaf holds only a few points. Once built, a single nearest-neighbor query first **descends** to the leaf containing the query point (this leg needs only $O(\log N)$ comparisons), obtains a batch of candidate neighbors, then **backtracks** upward: at each split node it checks whether "the distance from the query point to the splitting hyperplane" is still smaller than the current known k-th nearest — if not, the entire subtree on that side of the hyperplane cannot possibly contain a nearer point, and is **pruned away wholesale**, never to be entered. It is precisely this pruning that brings the average query complexity down to $O(\log N)$. A k-nearest-neighbor query maintains the current nearest k with a max-heap of size k; a radius search instead collects all points falling within a given radius, with the pruning criterion on backtracking switched from "the k-th nearest distance" to "the radius."

This chapter's project implements an implicit array-based kd-tree (using `std::nth_element` for the median split, with the axis rotating through $x/y/z$) and times it against brute force. For 20200 points doing **10000 queries of 8 nearest neighbors**, the kd-tree takes about **60 ms, brute force about 870 ms, a speedup of about 14.5×**; moreover the two results agree bit for bit — the kd-tree is not approximate and loses no precision; what it returns is exactly the nearest neighbors, only it cleverly avoids the impossible regions. This 14.5× shows little on a single query, but placed inside an iterative algorithm that queries nearest neighbors over and over (such as ICP, which every round finds correspondences for tens of thousands of points), it is the difference between a few minutes and tens of milliseconds — the engineering value of $\log N$ versus N is cashed out precisely in such inner loops that are "called ten million times."

The logarithmic complexity of the kd-tree has a precondition: the dimension must not be too high. When the dimension climbs to dozens or hundreds, "the distance to the hyperplane is almost always smaller than the distance to the nearest point," the pruning almost entirely fails, and the kd-tree degenerates back to near-brute-force — this is the **curse of dimensionality**. Fortunately, a three-dimensional point cloud has only 3 dimensions, and the kd-tree is in its element here. High-dimensional feature matching turns instead to approximate nearest neighbors or locality-sensitive hashing.

37.3 Noise and Outliers

The "dirt" in three-dimensional data divides into two fundamentally different kinds, and lumping them together leads you to reach for the wrong tool.

The first kind is **measurement noise**: every real surface point has a small random jitter along the measurement direction (usually the surface normal), which in this chapter's synthetic data is Gaussian noise of $\sigma = 0.10$ mm. Its hallmark is **small amplitude, zero mean, hugging the surface** — the points in a neighborhood are packed shoulder to shoulder, with uniform density. This kind of noise is handled by smoothing filters (later chapters of Part IX) and should not be dealt with by "deleting points."

The second kind is **outliers**, that is, flyers: arc light, specular reflection, multipath, and sensor artifacts conjure up a batch of points that lie on no real surface at all, hanging in mid-air, far from all normal structure. This chapter's 200 flyers are exactly so, sprinkled through the empty region $z \in [6, 44]$ mm. Their hallmark is just the opposite, **far from the surface, with an empty neighborhood** — a flyer has almost no other points around it, separated from its nearest neighbor by a large gap.

This "empty neighborhood" hallmark is exactly what can be used to pick out the flyers, and this is **statistical outlier removal (SOR)**. The algorithm is naive to the point of being blunt: for each point, use the kd-tree to find its k nearest neighbors, and compute its average distance d_i to these k neighbors; normal surface points are packed together, so d_i is small, while a flyer hangs alone in the air, so d_i is large. Treat the $\{d_i\}$ of all points as a distribution, find its mean μ and standard deviation σ, and judge every point with $d_i > \mu + t\sigma$ to be an outlier and remove it. This is really the 3σ rule from Chapter 2 applied to the quantity "neighborhood average distance," and it shares its roots with the statistical anomaly judgment of Chapter 26 — rather than presetting a fixed distance threshold, it lets the data's own distribution set the threshold.

Why should the threshold be set by the distribution rather than slapped down as a fixed number of millimeters? Because "how far counts as far" depends entirely on the density of the point cloud. The same 0.5 mm neighborhood-average distance is an outlier in a dense scan but normal in a sparse one. $\mu + t\sigma$ hands the scale over to the data itself, so switching to a different point cloud requires no retuning of parameters — and this is exactly the

This chapter takes $k = 8$, $t = 2$. On this data, the $\mu + 2\sigma$ of the neighborhood-average distance is 3.23 mm, by which **190 points are removed, of which all 190 hit true flyers (95% recall against the 200 flyers), with 0 false deletions** — not a single real surface point wronged. Figure 37.4 marks the 190 removed points with red diamonds; they fall neatly in the empty region, exactly those floating flyers.

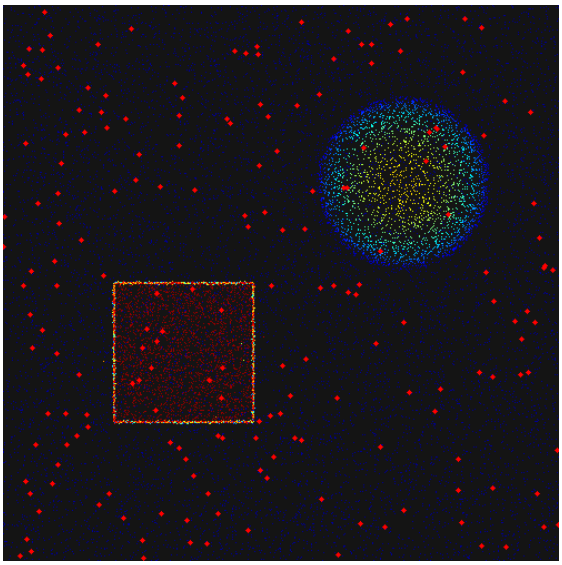


Figure 37.4: The SOR cleaning result (top view): red diamonds mark the 190 removed points, all floating flyers, with not a single real surface point falsely deleted. Compared with Figure 37.1, the sparse stray dots scattered across the figure are picked out precisely, while the surface points of the ground, cube, and spherical cap remain intact.

Where did the 10 missed flyers go? They were not removed because, unluckily (or rather too luckily), they happened to land near some real surface — if a flyer happens to drift very close above the ground or the cube, then real surface points slip in among its 8 nearest neighbors, its neighborhood-average distance is pulled down below the threshold, and it escapes the statistics. This is not a bug in the algorithm but an inherent boundary of the statistical

method: SOR judges "neighborhood density anomaly," and an outlier disguised as normal density it cannot tell apart. Being honest about these 10 escapees is more credible than reporting a 100% recall — downstream algorithms (such as robust fitting and ICP) ought to be resistant to the residual handful of outliers in the first place, rather than counting on the preprocessing to clean everything out in one pass.

37.4 SciVision Implementation

The I/O of the data forms is done with `SciPointCloud`, but there is a pitfall here that must be written down faithfully. `SciPointCloud` accepts a `SciVector3dArray` as its point set, and the constructor `SciVector3dArray(float*, size)` — the seemingly convenient "construct in one shot from a contiguous array" — is **inert** on this machine: after construction `Length()` is still 0, the points never went in at all. The reliable approach is to `Append` point by point:

```
SciVector3dArray pts;
for (int i = 0; i < n; ++i) {
    SciVector3d v(xyz[3*i], xyz[3*i+1], xyz[3*i+2]);
    pts.Append(v);              // append point by point; the float* constructor does not popul
}
pc.SetPoints(pts);             // pc.Length() = n thereafter
```

For nearest-neighbor queries, the SDK's `Sci3DKdTree` is **usable**, and its results agree with the hand-written kd-tree:

```
SCIMV::Sci3DKdTree kt;
long rc = kt.CreateKdTree(pc);
SciVector3d q(xyz[0], xyz[1], xyz[2]);
int num = 0; SciVector3dArray pos; SciIntArray ind; SciFloatArray dist;
kt.FindKNearestNeighbors(q, 8, &num, &pos, &ind, &dist);
// first four entries of dist = 0.000 / 0.516 / 0.802 / 0.999, bit-identical to the hand-writte
```

The first neighbor distance is 0.000 (the query point found itself), and the rest match the hand-written implementation exactly, which gives us a cross-check: this chapter's timing

and SOR are both based on the hand-written kd-tree (for determinism and timability), and the SDK's `Sci3DKdTree` serves as collateral evidence confirming correctness.

Data conversion `SciSv3DDataConvert`: the forward `ConvertPointCloudToRangeImage` (point cloud to range image) is usable, but the reverse `ConvertRangeImageToPointCloud` is inert on this machine (the surviving point count is 0), so the 2.5D loss experiment of Section 37.1 is done with hand-written binning, deterministic and controllable.

As for SOR, the `SORFILTER` feature of the SDK's `SciSv3DClean` **crashes** on this machine (the measured exit code drifts with the heap layout; both `0xC0000005` access violation and `0xC0000409` stack guard have been seen). This chapter therefore uses a separate subprocess probe to poke at it (a crash will not drag down the main flow), while the real SOR uses a hand-written implementation:

```
// hand-written SOR: each point's average distance to its k-neighborhood, removed at the  +2  t
for (int i = 0; i < N; ++i) {
    kd.knn(&xyz[3*i], K+1, &nb);          // K+1: includes itself
    double sd = 0; int c = 0;
    for (auto& p : nb) {                  // skip itself, accumulate distances to K neighbors
        if (p.second == i) continue;
        sd += std::sqrt(p.first);
        if (++c == K) break;
    }
    meanD[i] = sd / c;                     // neighborhood-average distance d_i
}
double mu = /* mean(meanD) */, sigma = /* std(meanD) */;
double thr = mu + 2.0 * sigma;            // = 3.23 mm
for (int i = 0; i < N; ++i)
    if (meanD[i] > thr) removed.push_back(i);   // remove 190 points
```

This division of labor — "the SDK does I/O and cross-checking, the core algorithm is hand-written" — is the norm for the 3D modules of Part IX: point-cloud-class algorithms (kd-tree, SOR, ICP, PCA) are mostly self-contained and easy to write by hand, and the

hand-written version further guarantees determinism and timability, whereas the maturity of the SDK's 3D modules on this machine is uneven — use it where it works, isolate it where it crashes. The complete runnable project is in `code/point_cloud_basics/`.

Industry Case: A Flyer Storm in Laser-Scanned Point Clouds

Laser scanning of weld seams is a disaster zone for flyers. Arc light, spatter, and specular reflection off the base metal fill the sensor with spurious echoes, and flyers are dense; if you take such a point cloud straight to measure the weld reinforcement height or the undercut depth, the flyers pull the measurement datum off as a whole. The correct flow is to do SOR cleaning first, then measure, after which the weld contour immediately settles down. But SOR is a double-edged sword: set the threshold too strict, and it will falsely delete real thin edges and sharp features along with the flyers — the burr tip of a weld seam and the deepest point of an undercut are themselves neighborhood-sparse "lone points," statistically looking much like flyers, and over-cleaning erases the very defect signal you set out to measure. The threshold must be set by the point cloud's own **noise distribution** ($\mu + t\sigma$ rather than a fixed number of millimeters). The rule: better to under-clean and hand the burden of outlier resistance to a downstream robust algorithm than to over-clean and throw away a real signal as noise.

37.5 Summary

- **Three-dimensional data takes three forms: the point cloud is most general, the range image most regular, the mesh manages connectivity.** The point cloud is an unordered point set that can represent arbitrary topology but needs an extra index; the range image is a regular 2.5D grid where two-dimensional operators apply directly, but each (x, y) stores only one z.
- **Point cloud to range image is lossy, and the loss is proportional to the scene's "verticality."** This chapter's 20200 points reprojected onto a 200×200 grid leave

only 12585 (losing 37.7%), with the vertical sidewalls and occluded regions vanishing as whole swaths — this is the essential limitation of single-viewpoint 2.5D, echoing the single-viewpoint acquisition of laser triangulation. If you can use a point cloud, do not collapse to a range image prematurely.

- **The kd-tree brings the nearest-neighbor query on an unordered point cloud from $O(N)$ down to $O(\log N)$:** build the tree by axis-rotating median splits, and on backtracking prune by "distance to the hyperplane." This chapter's 10000 queries of 8-NN take about 60 ms for the kd-tree and about 870 ms for brute force, a 14.5× speedup with bit-for-bit exact results — the bedrock of every iterative 3D algorithm (registration, filtering, features).
- **Three-dimensional "dirt" comes in two kinds with different remedies:** measurement noise is small-amplitude, zero-mean, hugging the surface, handled by smoothing; outlier flyers are far from the surface with an empty neighborhood, handled by deleting points via statistical outlier removal (SOR). Do not lump the two together.
- **SOR picks out flyers with a $\mu + t\sigma$ threshold on the "neighborhood average distance," with the threshold tracking the distribution:** this chapter's $k = 8$, $\mu + 2\sigma = 3.23$ mm removes 190 points, hits true flyers at 95%, with 0 false deletions. The 10 that escaped landed near a surface, disguised as normal density — an inherent boundary of the statistical method, so downstream should retain robustness against residual outliers.

For a more systematic treatment of point-cloud data structures, spatial indexing, and three-dimensional filtering, read further in the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018); for a quick reference on 3D geometry and range-image processing in a vision context, see (Szeliski 2022). The kd-tree spatial index originates in Bentley's classic paper introducing the multidimensional binary search tree (Bentley 1975), which brings the nearest-neighbor query of this chapter from $O(N)$ down to logarithmic time; and the point-cloud

operators used here and in later chapters — voxel downsampling, statistical outlier removal, normal estimation — are in practice mostly referenced to the Point Cloud Library (PCL) implementation, surveyed by Rusu and Cousins (Rusu and Cousins 2011). The subsequent Chapter 38 and Chapter 39 will develop, respectively, three-dimensional filtering/downsampling and point-cloud registration.

38 Point Cloud Preprocessing

Point a line-laser profilometer or a structured-light camera at a metal workpiece, press the capture button, and what you get back is never a clean, tidy 3D surface. The raw data carries three kinds of ailments mixed together: the height value jitters randomly at every sample point (the same origin as the noise in 2D images, except this time it is Z that jitters rather than gray value); scattered across the surface are patches of "black holes" — where the laser hits a steep wall, a deep bore, or a specular reflection and cannot be returned, those sample points get no height at all and become invalid pixels; and occasionally a few lonely "spikes" pop up, hundreds of microns above the true surface, the artifacts left by secondary reflections or anomalous matching. At the same time, a single range image easily contains hundreds of thousands to millions of points, and feeding it straight into registration or matching makes the computation explode in an instant. Noisy, full of holes, and far too dense — this is the raw stock that downstream measurement and inspection must confront.

Point cloud preprocessing is the very first procedure before measurement. What it does corresponds one to one with the 2D image enhancement of Part III: denoising, gap filling, downsampling, segmentation — only the battlefield has moved from the gray image to the 3D surface. This chapter can be read as the 3D counterpart of 2D enhancement.

Figure 38.1 is the synthetic scene that runs through this entire chapter: a $320\times240 = 76800$-pixel range image, with a base height of 1000 m and a gentle slope along the x direction, carrying a +400 m boss and a −300 m groove; onto it we have superimposed Gaussian height noise of $= 5$ m, 5% invalid pixels (3858 invalid points in total, appearing as black spots in the figure), and then sprinkled 245 spikes

(appearing as white spots). This single image gathers all three typical defects at once, and every section below starts from it.

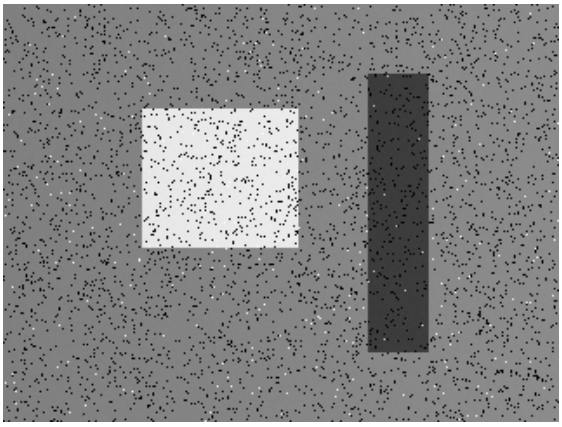


Figure 38.1: Raw synthetic range image (2.5D height map). Brightness encodes height: the base in the upper left is darker, the boss (central rectangle) is brighter, and the groove (right-hand vertical stripe) is darkest; the scattered black spots are the 5% invalid pixels, and the white spots are the 245 spikes rising above the true surface.

38.1 Range Image vs. Point Cloud: Where to Do It

Three-dimensional data can be stored in two ways, and preprocessing can be performed on either, but the cost and reusability differ greatly.

The first is the **range image**, also called the depth map or 2.5D height map: it is essentially an image on a regular grid, where each pixel stores one height value (Z), and the horizontal and vertical coordinates are implied by the pixel position (Chapter 37). Its greatest benefit is "regularity" — all the 2D algorithms based on pixel neighborhoods in Part III, namely median filtering, Gaussian filtering, morphology, and threshold segmentation, can be transplanted almost unchanged, simply swapping "gray value" for "height." The

second is the unordered **point cloud**: a pile of discrete (X, Y, Z) triplets that attach to no grid, able to faithfully express arbitrary topology (including overhangs, sidewalls, and multilayer structures that a range image cannot represent), but at the price of relying on a kd-tree for neighborhood queries, with every step running noticeably slower than on a regular grid.

For exactly this reason, **most SDK 3D preprocessing operations are done on the range image**, and this chapter's main arena is set here too. But there is one pitfall that does not exist in the 2D world and must be made clear first: **invalid pixels must not enter the computation as 0**. In a range image, invalid points are often marked with a count of 0 (or some sentinel value), meaning "there is no data here," not "the height here is 0." If a filter treats this as a height value of 0 and folds it into an average, then the true height next to a hole gets dragged sharply downward by this false 0, and denoising ends up manufacturing a fake trough. Therefore every step of 3D filtering and morphology must be **NaN-aware**: only valid neighbors are counted, and invalid points neither contribute nor get contaminated. Every hand-written operator in this chapter obeys this rule, and we will see its effect again and again below.

Rule of thumb: for single-view, single-layer surfaces (which describes the vast majority of inline inspection), prefer processing on the range image — it lets you directly reuse the mature 2D algorithms, runs fast, and saves memory; only when the object has overhangs, requires multi-view stitching, or has topology too complex for a grid to express do you enter the world of unordered point clouds (Chapter 39).

38.2 3D Filtering

Denoising is the first step of preprocessing, and the idea is entirely consistent with Chapter 6: on the height map, take a neighborhood around each point and perform an appropriate "averaging" to suppress the random jitter. We let two classic filters compete on the range image — a 3×3 **median filter** and a 3×3 **Gaussian filter**, both implemented as NaN-aware versions: if the center is invalid it stays invalid; if the center is valid it takes the median of only the valid heights in the neighborhood (median filter) or their weighted average (Gaussian filter).

To measure the denoising effect, we look at the root-mean-square (RMS, in m) of the flat base subregion's

height relative to the ground-truth plane. The results are as follows: the raw data has an RMS as high as **32.09 m** — note that this number is far larger than the 5 m noise standard deviation, because it is heavily polluted by the 245 spikes. The median filter pushes the RMS down to **2.28 m**, the Gaussian filter only to **12.08 m**, while median-then-Gaussian (median to remove spikes, Gaussian to smooth random noise) achieves **1.61 m**, the best of the field.

The root of the gap lies in the spikes. The lesson from Chapter 6 — **the median removes outliers, the Gaussian smears them** — replays unchanged here in 3D. Counting the residual spikes (valid points with |measured height − ground truth| > 200 m): 245 originally, **only 5 remain after the median filter**, while **14 still remain after the Gaussian filter**. The median filter treats each spike as a local outlier; after sorting it gets pushed to the tail of the list, never gets a turn to be the output, and is thus cleanly eliminated. The Gaussian filter, however, spreads the spike's few-hundred-micron height across its surrounding neighbors, and the result is not the elimination of the spike but smearing it into a shorter, fatter "bump" — the spike's energy has not vanished, it has merely been painted out. Figure 38.2 places the three side by side, and the eye can plainly see that the white spikes in the raw image have completely disappeared in the median result, but have turned into patches of pale halos in the Gaussian result.

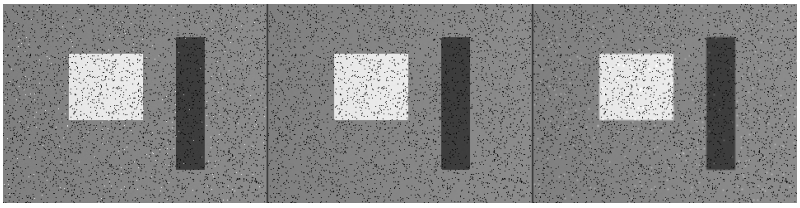


Figure 38.2: Three-panel 3D filtering comparison: raw (left, white spots are spikes) | median (center, spikes removed, edges sharp) | Gaussian (right, spikes smeared into pale bumps, boss edges slightly blurred). On the same data, the opposition of "median removes spikes, Gaussian smears spikes" replays on the 3D height map.

Therefore the final output takes the **median + Gaussian** cascade (Figure 38.3): the first median pass clears away the spikes and any impulse that may have slipped through, and the second Gaussian pass further soothes the residual random jitter. This combination preserves the edges of the boss and groove while pressing the flat-region RMS below 2 m, laying a clean foundation for the measurement to come.

This section uses isotropic Gaussian/median filters. If you want both to suppress noise and to preserve the steep walls of the boss and groove, you can switch to an **edge-preserving** 3D filter — the bilateral filter or the guided filter — which introduces the "only participate in the average if the heights are close" criterion into the weighting, with a mechanism stemming from the same lineage as the bilateral filter of Chapter 6.

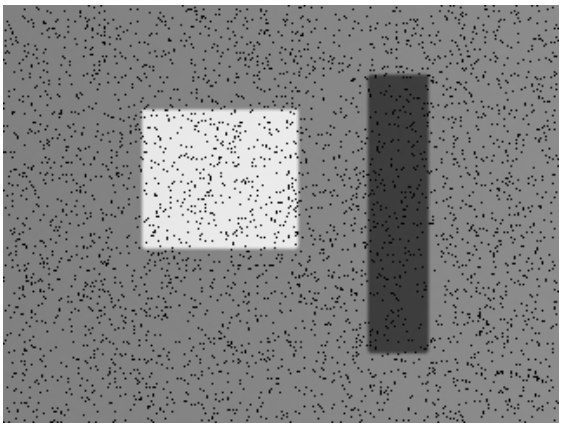


Figure 38.3: Range image after the median + Gaussian cascade filter: the spikes are removed, the random noise is soothed, and the boss and groove edges stay sharp; the invalid pixels (black spots) are not yet handled, left to the morphology of the next section.

38.3 3D Morphology: Hole Filling

Filtering dealt with the "jitter" and the "spikes," but did not touch the 3858 black holes — the invalid pixels are still holes. Hole filling is a job for morphology, and the idea is consistent with the 2D morphology of Chapter 8: use a **closing operation (dilation followed by erosion)** to fill in small-sized holes. On the height map, for each invalid pixel we inspect its 8-neighborhood, and if the number of valid neighbors reaches a threshold, we fill it with the median of those valid neighbors, iterating for several rounds — this is equivalent to letting the valid region "grow" inward a few

rings, gradually closing up the isolated small holes while leaving the large continuous gaps untouched.

The experimental result is crisp and clean: the invalid pixels drop from **3858 to 1** — the closing operation filled 3857 small holes (Figure 38.4). Most of the 5% invalid pixels are scattered holes of 1 to 3 pixels, and a few rounds of closing can gather them all up.

Figure 38.4: Range image after morphological closing for hole filling: the originally scattered black invalid pixels (3858 of them) are almost all filled with the median of valid neighbors (1 remaining), and the surface regains continuity.

But hole filling must be done with vigilance. **Small holes can be filled, but what you fill a big hole with is fake data.** Closing fills a hole by interpolating from the surrounding valid points, which carries the implicit assumption that "the surface beneath this hole is continuous with its surroundings." For scattered holes of a few pixels, this assumption essentially holds, and the filled height is credible enough; but once you face a large continuous gap — say, a blind zone left where the laser was wholly occluded by a deep groove or steep wall — forcibly interpolating from the surrounding heights amounts to fabricating, out of thin air, a stretch of surface that was never measured at all. This is the same reasoning as the "better missing than fabricated" dilemma in Chapter 33: the missing data itself often carries

Why "closing" and not "opening"? The closing operation fills the dark holes inside the foreground (the valid region), which is exactly the invalid pixels we want to fill; the opening operation, by contrast, erases small bright spots, a use of an entirely different kind. The choice of morphological operator always depends on which — the "hole" or the "spike" — is the object you want to eliminate.

information (there is a deep structure here, there is an abnormal reflection here), and mindlessly filling it flat instead erases the genuine signal. So the closing operation in this section deliberately sets the constraint that it "fills only when the number of valid neighbors reaches a threshold," letting it fill only small holes and pass over large ones — and that 1 remaining invalid point is precisely the embodiment of this restraint.

38.4 3D Sampling

After denoising and hole filling, the surface is clean, but there are still too many points. This section addresses the "too dense" problem. **Voxel grid downsampling** is the most commonly used means of point cloud downsampling: cut 3D space into small cubes of fixed edge length (voxels), replace all points that fall into the same voxel with one representative point (usually the centroid), and the point density is thereby uniformly thinned out. Its benefit is "spatial uniformity" — no matter where the original point cloud is dense and where it is sparse, the output point spacing is uniformly constrained by the voxel edge length, with no local over-density.

Taking a voxel edge length of 0.4 mm, this chapter's range image goes from **76799 valid points down to 4901, a 15.7× reduction** (Figure 38.5). The key is that this reduction barely harms the structure: the densely packed points on the flat base already have similar heights and fall into the same batch of voxels, so collapsing them into representative points loses almost no information; while the edges of the boss and the groove, because their heights span different Z voxel layers, keep their points, and the edge density is relatively prominent by comparison. Figure 38.5 clearly shows this: the flat region is sparse in points, yet the contour lines of the groove and the boss remain dense — the structural information has been preserved intact.

Why must 3D always be downsampled? Because downstream algorithms are extremely sensitive to point count. The ICP in registration (Chapter 39) must search for the nearest neighbor

Voxel sampling is not the only kind of downsampling. **Uniform sampling** takes every n-th point at a fixed stride, simple but liable to miss details; **voxel sampling** takes a representative point per spatial cell, with uniform density; **curvature-adaptive sampling** thins aggressively in flat regions and keeps more in high-curvature regions (edges, corners), saving the most points while best preserving structure, at the cost of having to first estimate normals and curvature. The three are chosen according to "whether you need to preserve

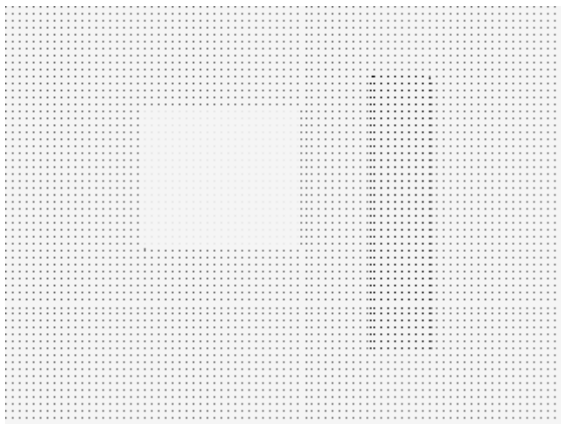


Figure 38.5: Voxel grid downsampling result (voxel edge length 0.4 mm): 76799 points reduced to 4901 (15.7×). The flat region is sparse in points, the structural edges (groove and boss contours) are dense in points, and the downsampling preserves the structure even as it greatly reduces the count.

of every point in every round; running ICP or feature matching on a million-point cloud means a million kd-tree queries per single round, and after dozens of rounds the computation flat-out explodes; template matching and surface registration are the same. Cutting the point count by an order of magnitude or two first, before doing this heavy work, is often the key step that turns "won't run" into "real-time." Of course downsampling is also scene-dependent: for localization and registration, the downsampled result is both fast and stable; but for high-precision volume or dimensional measurement, you should use full resolution — a trade-off we will encounter again in the next section's industry case.

38.5 3D Thresholding and Segmentation

The last step of preprocessing is often to separate the "region of interest" from the background and hand it off to subsequent measurement. The simplest and most direct 3D segmentation is **height thresholding**, an idea exactly like the 2D gray-value thresholding of Chapter 7: set a height

threshold, classify everything above it as the target and everything below it as background. This chapter separates the boss from the base by "height $>$ 1200 m" — the base sits around 1000 m, the boss rises 400 m to about 1400 m, and the 1200 m line falls cleanly between the two.

The result is nearly perfect: the segmentation yields **7197 pixels**, while the boss ground truth (GT) is **7200 pixels**, with **0 false positives (FP) and 3 false negatives (FN)**, and an intersection-over-union of **IoU = 0.9996** (Figure 38.6). All 3 false negatives are on the boss edge — where the height happens to sit right at the threshold and, after being slightly smoothed by filtering, dipped below the line, an acceptable boundary effect. Achieving this kind of precision owes much to the groundwork of the previous sections: filtering first removed the spikes that would have caused massive misclassification, and hole filling then filled the holes inside the boss that would otherwise have been judged as background, so that threshold segmentation could come out this clean.

Figure 38.6: Height threshold segmentation result: the boss separated by height $>$ 1200 m is highlighted in green overlaid on the gray-value height base map. Segmentation 7197 pixels vs. ground truth 7200, FP = 0, FN = 3, IoU = 0.9996.

Height thresholding is only the starting point of 3D segmentation. When the height difference between target and

background is not obvious, or when the regions to be separated are coplanar, you need more advanced means: **region growing** clusters adjacent points into patches by the continuity of normals or curvature, and **plane segmentation (RANSAC plane fitting)** directly extracts plane primitives one by one from the point cloud (echoing the primitive fitting of Chapter 41). But for the most common industrial targets of the "boss that rises a notch / groove that drops a notch" kind, a single height threshold is often enough — simple and reliable.

38.6 SciVision Implementation

The usability of this chapter's corresponding SciVision 3D module varies greatly in testing on this machine; it is recorded faithfully below for engineering trade-offs.

- **SciSv3DFilter**'s `Median` / `Gaussian` are **usable** and NaN-aware (they correctly skip invalid pixels), consistent with the hand-written implementations;
- **SciSv3DMorphology**'s `Close` (closing for hole filling) is **usable**;
- **SciSv3DThreshold** is **inert** on this machine — `ManualThreshold` returns rc = 0 but produces empty output, so segmentation has to be done yourself;
- **SciSv3DSampling**'s `Sampling` overload on the range image **fails** (returns 123403001), and switching to the point cloud's `VoxelSampling` is **non-deterministic across runs** (the same input yields a drift of 4821/4821/0 over three runs), which is unreliable.

Therefore the main line of this chapter uses entirely hand-written, NaN-aware, deterministic implementations to produce the figures and numbers, while the various SDK 3D modules are placed in an isolated subprocess as probes, corroborating their return codes and usability (the 3D modules on this machine occasionally throw a 0xC0000005 access violation, so they are isolated to avoid dragging down the main flow). Below are the two most critical hand-written snippets. The NaN-aware median filter keeps the center

invalid if it is invalid, otherwise takes the median of only the valid neighbors:

```
// 3x3 NaN-aware median: suppresses spikes, preserves edges; invalid points neither participate
for (int dy = -r; dy <= r; ++dy)
  for (int dx = -r; dx <= r; ++dx) {
    double v = h[(yy)*W + xx];
    if (valid(v)) w.push_back(v);   // collect only valid neighbors
  }
std::sort(w.begin(), w.end());
out[y*W + x] = w[w.size() / 2];     // median
```

Voxel grid downsampling quantizes (X, Y, Z) to a voxel key and keeps one representative point per voxel:

```
long long ix = floor(X / L), iy = floor(Y / L), iz = floor(Z / L);
long long key = (ix*100003LL + iy)*100003LL + iz;   // voxel key
Vox& g = grid[key];                                 // merge points falling into the same voxel
if (g.n == 0) { g.rx = x; g.ry = y; }               // first point as representative
g.sx += X; g.sy += Y; g.sz += Z; ++g.n;
```

The complete runnable project is located at `code/point_cloud_preprocessing/`; readers can change the noise level, voxel edge length, and segmentation threshold to reproduce all the figures and numbers themselves.

Industry Case: Holes and Downsampling in 3D Solder Paste Inspection

SPI (Solder Paste Inspection) uses a 3D camera to measure the volume of solder paste on each pad. The specular reflection of solder paste prevents the laser from being returned, leaving a large number of invalid pixels — and if you integrate the data with holes directly, the holes are treated as zero height, the volume is systematically underestimated, and a flood of "insufficient paste" false alarms results. The correct approach is to first fill the measurement holes with morphological closing, then integrate. But hole filling has an iron rule: **you may only repair "measurement holes," never fill in "process holes"** — somewhere with no paste printed at all (a missed print) is a

genuine process defect and must never be filled by the hole-filling algorithm; distinguishing the two kinds of holes relies on a joint judgment of size, position, and surrounding morphology. On downsampling: measuring volume requires full resolution to preserve precision, while locating pad positions on a large board is fast and accurate enough with a downsampled point cloud. The lesson is just one sentence: at every step of preprocessing, you must distinguish "repairing a measurement defect" from "erasing a genuine signal."

38.7 Summary

The key points of this chapter can be distilled as follows.

- **Preprocessing is the 3D counterpart of 2D enhancement.** The four steps of denoising, hole filling, downsampling, and segmentation correspond respectively to spatial filtering, morphology, sampling, and thresholding — swap "gray value" for "height," and most of the mature 2D algorithms can be directly transplanted to the range image for reuse.
- **Invalid pixels must never enter the computation as 0.** Every operator step in 3D must be NaN-aware, counting only valid neighbors; otherwise a single hole will drag the surrounding true heights down into a fake trough.
- **The median removes spikes, the Gaussian smears them.** The lesson of Chapter 6 replays in 3D: the median presses the spikes down to just 5 (RMS 32.09 $\rightarrow$ 2.28 m), while the Gaussian paints the spikes into bumps (14 residual); the median + Gaussian cascade achieves 1.61 m.
- **Hole filling must distinguish measurement defects from genuine signals.** Closing is very effective at filling small holes (3858 $\rightarrow$ 1), but what forced interpolation fills a large gap with is fake data — the same dilemma as "better missing than fabricated" at occlusions.
- **Downsampling is the prerequisite for 3D to run.** The voxel grid reduces 76799 points to 4901 (15.7$\times$) while

preserving the structural edges; ICP/matching on a million points will explode without downsampling first, but high-precision measurement still needs full resolution.

For a more systematic treatment of filtering, morphology, and segmentation of 3D data in industrial inspection, see further the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018). This chapter's statistical outlier removal (SOR) with its "neighborhood average distance $\mu + t\sigma$" criterion comes from the denoising pipeline Rusu et al. proposed while building point-cloud object maps for household scenes (Rusu et al. 2008); the open-source reference implementation of voxel-grid downsampling and the other operators above is surveyed in the Point Cloud Library (PCL) overview (Rusu and Cousins 2011).

39 ICP Registration and 3D Rectification

Scan the same workpiece once from each of two viewpoints and you get two point clouds; to stitch them into a complete surface, you first have to align them. To compare a scanned physical object against its designed CAD model and judge whether assembly is correct, you first have to "place" the measured cloud into the model's coordinate frame. Before a robot can grasp a part from a bin, it first has to know that part's precise pose relative to the gripper — three seemingly unrelated tasks that share one and the same skeleton: **rigidly transform one point cloud onto another so that the two coincide as closely as possible**. This is three-dimensional **registration**, and ICP (iterative closest point) is the workhorse algorithm that carries registration on the industrial floor.

This chapter runs all of its experiments on a single synthetic scene (Figure 39.1): a **model cloud (model)** — a square box and a spherical cap sitting on a conveyor plane, 5678 points in all; and a **scan cloud (scan)** — obtained by applying a known rigid transform to the model (rotation of 12° about the axis $(1, 2, 2)$, translation $(5, -3, 2)$ mm), adding measurement noise with $\sigma = 0.1$ mm, then cropping away one side and patching in, on the right, an "intruder wall" that does not exist in the model at all — 6497 points in all. Only 4310 points of the two clouds truly correspond; another 2187 points (**33.7%**) are non-overlapping with no correspondence. In the figure the model is blue and the scan is red, clearly misaligned at the start; that isolated red vertical wall on the right is the intruder structure — it has no corresponding point in the model, yet the algorithm must identify and exclude it. This "not fully overlapping" setup is not self-sabotage: in real

scans, non-overlap from the camera's field-of-view boundary, neighboring parts, and bin walls is almost the norm.

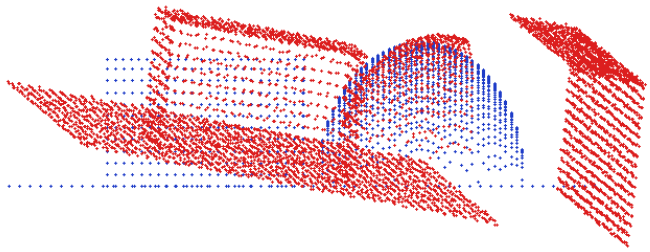


Figure 39.1: Initial poses of the model cloud (blue) and scan cloud (red), XZ orthographic side view. The two clouds are clearly misaligned; the isolated red vertical wall on the right is an intruder structure present only in the scan (an abstraction of a bin wall), with no corresponding point in the model.

39.1 The Registration Problem and ICP

What rigid registration must solve for is a **rotation matrix** $\mathbf{R}$ (3×3, orthogonal with $\det \mathbf{R} = 1$) and a **translation vector** $\mathbf{t}$, such that transforming each point $\mathbf{p}_i$ of the source cloud makes it coincide with the target cloud: $\mathbf{R}\mathbf{p}_i + \mathbf{t} \approx \mathbf{q}_i$. This is the same problem as solving for the planar pose (x, y, θ) in 2D position correction in Chapter 19, upgraded to three dimensions — 2D has 3 degrees of freedom, a 3D rigid transform has 6 (3 rotational + 3 translational).

The difficulty is that we do not know which point of the source cloud corresponds to which point of the target cloud. The **correspondence** and the **transform** are each other's cause and effect — knowing the correspondence lets you solve the transform, and knowing the transform lets you find the correspondence. ICP breaks this chicken-and-egg deadlock

with a naive but effective iteration, doing three things each round:

1. **Find correspondences by nearest neighbor**: transform the source cloud by the current estimate $(\mathbf{R}, \mathbf{t})$, then for each source point find its **nearest-neighbor correspondence** in the target cloud. This step is ICP's computational bottleneck and must use a **kd-tree** to reduce the brute-force $O(NM)$ nearest-neighbor query to $O(N \log M)$ (Chapter 37).
2. **Reject far correspondences**: a nearest neighbor is not guaranteed to be a "true" correspondence — a point on the intruder wall also gets assigned a closest model point, but that correspondence is false. Rejecting outlier pairs by correspondence distance is ICP's key gate for handling non-overlap. This chapter uses the median method: take the median $\tilde{d}$ of all correspondence distances and keep correspondences with distance $\leq 2.5\,\tilde{d}$, shutting out large-distance correspondences such as the intruder wall.
3. **Solve for the optimal rigid transform**: on the filtered correspondences, solve for the rigid transform that minimizes the sum of squared distances between corresponding point pairs. This step has a closed-form solution, namely the **Kabsch / Umeyama algorithm**.

The mathematics of the third step is worth unpacking; it is a beautiful application of the SVD from Chapter 2. Suppose after filtering there are n correspondence pairs $\{(\mathbf{p}_i, \mathbf{q}_i)\}$, and we want to minimize

$$E(\mathbf{R}, \mathbf{t}) = \sum_{i=1}^{n} \left\| \mathbf{R}\mathbf{p}_i + \mathbf{t} - \mathbf{q}_i \right\|^2.$$

First subtract the centroid from each of the two point sets: $\bar{\mathbf{p}} = \frac{1}{n}\sum_i \mathbf{p}_i$, $\bar{\mathbf{q}} = \frac{1}{n}\sum_i \mathbf{q}_i$, and let $\mathbf{p}'_i = \mathbf{p}_i - \bar{\mathbf{p}}$, $\mathbf{q}'_i = \mathbf{q}_i - \bar{\mathbf{q}}$. One can prove that the optimal translation aligns the two centroids, while the rotation depends only on the centered points. Construct the 3×3 **cross-covariance matrix**

$$\mathbf{H} = \sum_{i=1}^{n} \mathbf{p}'_i \, \mathbf{q}'^{\top}_i,$$

take its singular value decomposition $\mathbf{H} = \mathbf{U}\Sigma\mathbf{V}^{\top}$, and then the optimal rotation and translation are

$$\mathbf{R} = \mathbf{V}\,\mathrm{diag}(1, 1, d)\,\mathbf{U}^{\top}, \quad d = \mathrm{sign}\big(\det(\mathbf{V}\mathbf{U}^{\top})\big), \qquad \mathbf{t} = \bar{\mathbf{q}} - \mathbf{R}\,\bar{\mathbf{p}}.$$

That middle $\mathrm{diag}(1, 1, d)$ is the crucial stroke: a plain $\mathbf{V}\mathbf{U}^{\top}$ may yield a "mirror" with determinant -1 (a reflection rather than a rotation), and flipping the last column with d forces it back into the legitimate rotation group $SO(3)$. Once $(\mathbf{R}, \mathbf{t})$ is found, compose it onto the current estimate and proceed to the next iteration, until convergence.

Why centering + SVD? Expand E, and after centering the translation term and the rotation term decouple: the optimal translation necessarily lands the transformed source centroid on the target centroid, and the remaining rotation term $\sum_i \|\mathbf{R}\mathbf{p}'_i - \mathbf{q}'_i\|^2$ is equivalent to maximizing $\mathrm{tr}(\mathbf{R}^{\top}\mathbf{H})$, which under the orthogonality constraint is given optimally in one step by the SVD of $\mathbf{H}$. This is precisely the direct realization of "SVD gives the optimal orthogonal approximation" from Chapter 2.

39.2 Convergence and Accuracy

Give ICP a **good initial value** — here simply the identity matrix (at synthesis time the rotation is only 12° and the translation a few millimeters, so the identity already lies within the convergence basin) — and run point-to-point ICP. The result: **convergence in 58 iterations, final correspondence RMS 0.6443 mm**, recovered rotation angle 10.69° (GT is 12°, rotation-angle error 1.97°), and translation error 0.233 mm. Figure 39.2 shows the red scan cloud snugly hugging the blue model after alignment, while that intruder wall on the right still stands alone where it was — correspondence rejection correctly judged it non-overlapping and kept it from contaminating the transform estimate.

What is worth pondering is this 0.6443 mm: it converged correctly (the pose is right) yet clearly stopped above the 0.1 mm noise floor. The reason hides in the cloud's geometry — this cloud is dominated by a large expanse of conveyor plane, and the **point-to-point** metric has a congenital

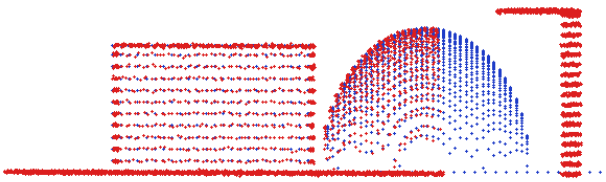


Figure 39.2: Point-to-point ICP after convergence (good initial value). The red scan cloud coincides with the blue model cloud; the intruder wall on the right is rejected for excessive correspondence distance and correctly took no part in the alignment.

weakness on planes: a source point sliding along the **tangential** direction of the target surface barely changes its distance to the nearest target point, so the objective is nearly flat in the tangential direction. Each iteration can therefore move only a little, and convergence is slow and prone to stalling in a shallow minimum. The red curve in Figure 39.3 depicts this plainly: after the RMS drops quickly, it drags a long, nearly flat tail, crawling slowly toward the plateau over 58 iterations rather than crisply slamming into the noise floor.

39.3 Local Minima: ICP's Achilles Heel

ICP only guarantees **local convergence** — it dutifully rolls down to the valley bottom nearest the initial value, but does not guarantee that this is the globally optimal valley bottom. To verify this, deliberately spoil the initial value: on top of the good initial value, add a further 40° rotation about the Z axis, leave everything else unchanged, and rerun the same point-to-point ICP. The result is startling: **the full 60 iterations are used up, with final RMS 2.0824 mm,**

Termination criterion: ICP usually stops when "the RMS change between two consecutive rounds falls below a threshold" or when "the maximum number of iterations is reached". This chapter's implementation stops when the RMS change is $< 10^{-5}$ mm, with a 60-iteration cap as a backstop. Too loose a criterion quits early with insufficient accuracy; too tight a one spins idly near the noise floor — the long tail of point-to-point on a plane-dominated cloud is exactly the kind of scene where the criterion is hard to tune.

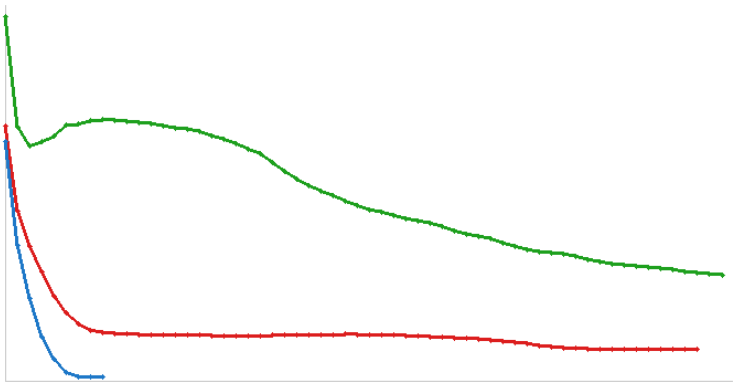


Figure 39.3: Three RMS convergence curves: red = point-to-point (good init), crawling along a long tail after dropping to a plateau; green = point-to-point (bad init), lingering high at a wrong minimum; blue = point-to-plane (good init), slamming fastest into a near-zero noise floor.

rotation-angle error 29.3°, and translation error 8.20 mm. The green "aligned" result in Figure 39.4 is offset from the model as a whole — ICP converged, but converged to a **wrong local minimum**. The green curve in Figure 39.3 lingers high the whole way and never comes back down: the nearest-neighbor correspondences are mismatched from the start, the wrong correspondences solve a wrong transform, the wrong transform reinforces the wrong correspondences, and it sinks ever deeper.

This leads to the first iron rule of ICP in engineering practice: **ICP is a refiner, not a searcher**. It can polish a "roughly correct" pose to sub-millimeter accuracy, but it is powerless to search out the correct answer from a pose that is "far off". Every local minimum has its own **convergence basin**; whichever basin the initial value falls into, ICP rolls toward that valley bottom; only when it falls into the basin of the global optimum is the result correct.

Industrial 3D localization therefore almost always adopts a **coarse-to-fine pipeline**: first use a **coarse registration**

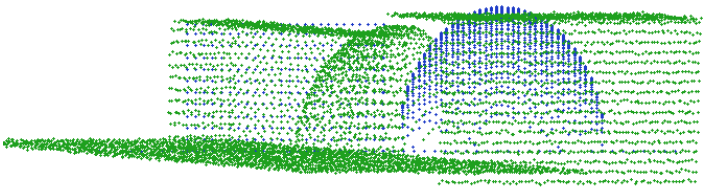


Figure 39.4: Under a bad initial value (an extra 40° rotation about Z), point-to-point ICP converges to a wrong local minimum. Green is the converged result, offset from the blue model as a whole; this is intuitive evidence that "ICP needs a good initial value".

algorithm that is insensitive to the initial pose — based on feature descriptors or point pair features (PPF) from Chapter 40 — to obtain a rough initial value that, though coarse, lands in the correct basin, then hand it to ICP for refinement. Coarse registration is responsible for "finding the right basin", ICP for "rolling to the bottom", each playing its part.

39.4 The Point-to-Plane Variant

The point-to-point weakness exposed in Section 39.2 — slow convergence because tangential sliding is unpenalized — has a tailored remedy: the **point-to-plane** variant. Instead of minimizing the straight-line distance from a source point to a target point, it minimizes the distance from a source point to the **tangent plane at the target point**. Let the surface normal at target point $\mathbf{q}_i$ be $\mathbf{n}_i$ (obtained by taking the k nearest neighbors of each point in the target cloud, doing PCA, and taking the eigenvector for the smallest eigenvalue), and each round solves

$$\min_{\mathbf{R},\mathbf{t}} \sum_i \Big((\mathbf{R}\mathbf{p}_i + \mathbf{t} - \mathbf{q}_i) \cdot \mathbf{n}_i \Big)^2 .$$

Penalizing only the deviation along the normal and letting tangential sliding pass — this is exactly what loosens the shackle that jams point-to-point. Each round linearizes the small-angle rotation as $\alpha = (\alpha_x, \alpha_y, \alpha_z)$, combines it with the translation into a 6-dimensional unknown, and solves a 6×6 normal equation.

The effect is immediate. With the same good initial value and the same data, the point-to-plane variant **converges in just 9 iterations to RMS 0.0991 mm**, with rotation-angle error 0.016° and translation error 0.005 mm — slamming straight into the 0.1 mm noise floor, whereas point-to-point takes 58 iterations and still stops at 0.6443 mm. The blue curve in Figure 39.3 plunges steeply and reaches bottom in a few steps, a sharp contrast to the red one's long tail.

An intuitive grasp of this 9-versus-58 gap: on a planar region, point-to-point is like making a source point "ice-skate" on the target surface, compacting only a little normal distance per step while the tangential freedom is wasted; point-to-plane hands the tangential degrees of freedom straight back to the optimizer, so every step is spent on the cutting edge (the normal). The cost is having to compute target-cloud normals and being sensitive to the quality of normal estimation. Precisely because it is faster and more accurate, **most industrial ICP implementations default to point-to-plane**, with point-to-point serving more as a teaching baseline and a fallback when normals are unreliable.

39.5 3D Rectification

A close relative of registration is **3D correction**: "straightening" a workpiece from the casually placed pose it happens to be in into a reference coordinate frame. Chapter 5 calibrates pixels to a physical coordinate frame; 3D correction aligns the measured workpiece pose to a design reference — before measuring flatness, checking tilt, or making subsequent measurements, you often first have to level the reference surface.

The most common kind is **plane correction**: a workpiece that should lie horizontal actually carries a little tilt, and needs to be rotated upright to the $z = 0$ reference plane. The procedure is direct — do a **least-squares plane fit** to the workpiece surface points to get the normal $\mathbf{n}$, then solve for a rotation matrix that rotates $\mathbf{n}$ to the $+Z$ axis, and apply it to the whole cloud. There is a ready-made construction for the rotation that "turns the unit vector $\mathbf{n}$ to $\mathbf{z}$": take the rotation

axis as $\mathbf{n} \times \mathbf{z}$ and the rotation angle as the angle between the two, and plug into the Rodrigues formula.

The experiment makes a plane with a 7° compound tilt (about a slightly oblique axis) plus noise. After fitting the normal and rotating upright, the residual tilt angle drops from 6.9999° **to** 0.0000° — one fit plus one rotation seats the workpiece flush. Figure 39.5 draws the before-and-after comparison in YZ side view: the red tilted point band is rotated upright into the green flat band that hugs the $z = 0$ reference line.

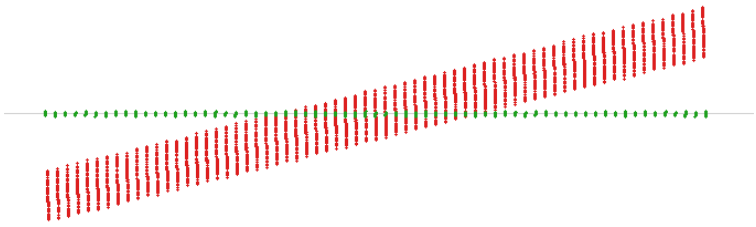


Figure 39.5: 3D plane correction, YZ orthographic side view. Red is the 7° tilted plane point band before correction; green is the result after fitting the normal and rotating upright to +Z, with residual tilt angle 0.0000°, hugging the z=0 reference line (gray).

Looking one layer deeper, **correction is registration against an "ideal pose"**: the target it aligns to is not another measured point cloud but an analytically defined reference (here the $z = 0$ plane). It therefore needs no nearest-neighbor iteration — the target geometry is closed-form, and a single fit yields the transform. Generalize this idea, and aligning to CAD reference planes, reference axes, and reference holes are all the same class of "align to ideal geometry" correction problems.

39.6 SciVision Implementation

The geometry and mathematics of this chapter — kd-tree, Kabsch (SVD), point-to-point / point-to-plane iteration, plane fitting and rotating upright — are **all hand-written**, and this is precisely the core of the chapter; the SciVision SDK here only plays the supporting role of point-cloud IO and cross-validation evidence. This choice has measured backing: several of this machine's SDK 3D entry points are simply unreliable in registration scenarios — some overloads of `Sci3DKdtree::CreateKdTree` return error code `121106105`, and `Sci3DAxisTransform::Transform3DPointArray` crashes outright with `0xC0000005`; only `SciSv3DSurfaceCorrection::ApplyPlaneCorrection` is usable, and this chapter uses it to corroborate the plane-correction result. The sample project runs each SDK probe in a separate subprocess to isolate potential crashes without affecting the main flow's figure output.

The main loop of nearest neighbor + correspondence rejection (median threshold):

```
for (size_t i = 0; i < src.size(); ++i) {
    cur[i] = mul(res.R, src[i]) + res.t;   // transform source point by current pose
    nn[i]  = kd.nearest(cur[i], d2[i]);    // hand-written kd-tree nearest neighbor
}
std::sort(sorted.begin(), sorted.end());
double med    = sorted[sorted.size() / 2];          // median of correspondence distances
double thresh = std::max(med * 2.5, 0.5);           // median-method rejection threshold
// keep only correspondences with dist <= thresh for Kabsch -- large-distance intruder-wall pa
```

The filtered correspondences are handed to Kabsch to solve the closed-form rigid transform (centering + cross-covariance SVD + determinant correction), corresponding line by line to the formulas in Section 39.1:

```
Mat3 H = /* Σ p_i' q_i'^T */;
symEig3(mul(transpose(H), H), w, V);                // eigendecomposition of H^T H -> right s
// columns of U = H v_i /  _i;  _i = sqrt( _i)
Mat3 VUt = mul(V, transpose(U));
```

```
double d  = det(VUt) < 0 ? -1.0 : 1.0;              // prevent reflection, force det(R)=+1
R = mul(mul(V, Mat3{{1,0,0, 0,1,0, 0,0,d}}), transpose(U));
t = ct - mul(R, cs);
```

The point-to-plane variant linearizes each correspondence into a 6×6 normal equation, with the unknowns being the small-angle rotation α and the translation:

```
double r = dot(s - q, nrm);          // signed distance from point to target tangent plane
Vec3   c = cross(s, nrm);            // Jacobian of the rotation part
double a[6] = {c.x, c.y, c.z, nrm.x, nrm.y, nrm.z};
for (int u = 0; u < 6; ++u) {        // accumulate the normal equations A x = b
    for (int v = 0; v < 6; ++v) A[u][v] += a[u] * a[v];
    bb[u] += -r * a[u];
}
```

Plane correction uses `fitPlane` (least-squares solution of $z = ax + by + c$) to find the normal, then `rotateVecToZ` (rotation axis $\mathbf{n} \times \mathbf{z}$ + Rodrigues) to rotate upright — two hand-written functions, with `SciSv3DSurfaceCorrection` as corroboration. The complete runnable project is in `code/icp_registration/`.

Industry Case: Point Cloud Registration for Robotic Picking

Parts to be picked lie scattered in a bin; the robot relies on a 3D camera to scan a point cloud, register it to the CAD model, solve the 6DoF pose, and then plan the grasp. An early approach used pure ICP directly, and the line crashed again and again — with the parts' initial poses in disarray, ICP kept falling into wrong local minima, the pose drifted off, and the manipulator grabbed off-target or even grabbed nothing. The key to the overhaul was to add the coarse-registration link: first use the point pair features of 3D matching (Chapter 40) to compute a rough pose, then hand it to ICP to refine to sub-millimeter accuracy, and only then did the pose stabilize. Another unavoidable pitfall is non-overlap: bin walls and neighboring parts both enter the field of view, and these points have no correspondence on the target model and must be blocked one by one by correspondence rejection (distance threshold), or they will drag the pose outward. The

lesson: **ICP is a refiner, not a searcher; there must always be a coarse registration in front to feed it an initial value.**

39.7 Summary

- **Registration = solving for the rigid transform $(\mathbf{R}, \mathbf{t})$ that makes two point clouds coincide**, the 3D upgrade of 2D position correction (Chapter 19); ICP uses the loop "find correspondences by nearest neighbor → reject far correspondences → solve the optimal transform by SVD → iterate" to break the deadlock in which correspondence and transform are each other's cause and effect.
- **Kabsch / Umeyama gives the closed-form optimal rigid transform**: subtract the centroid, construct the cross-covariance matrix $\mathbf{H}$, take its SVD, $\mathbf{R} = \mathbf{V}\,\mathrm{diag}(1, 1, d)\,\mathbf{U}^\top$, where d prevents degeneration into a reflection — this is a direct application of the SVD from Chapter 2.
- **ICP only guarantees local convergence and is highly sensitive to the initial value**: with a good initial value it converges in 58 iterations to RMS 0.6443 mm; with a bad initial value (an extra 40° rotation) it falls into a wrong local minimum, with rotation error 29.3°. Industry must go coarse-to-fine — first use 3D matching (Chapter 40) for coarse registration to supply the initial value, then use ICP to refine.
- **The point-to-plane variant is far faster than point-to-point**: penalizing only normal deviation and letting tangential sliding pass, this chapter's experiment reaches RMS 0.0991 mm in 9 iterations (point-to-point still stops at 0.6443 mm after 58), so industrial ICP mostly adopts point-to-plane.
- **3D correction is registration against an "ideal pose"**: fit a reference plane + rotate upright to $+Z$, dropping a 7° tilt to 0.0000° in one step; non-overlap (intruder walls, bin walls) must be blocked by correspondence rejection, a prerequisite for registration to land

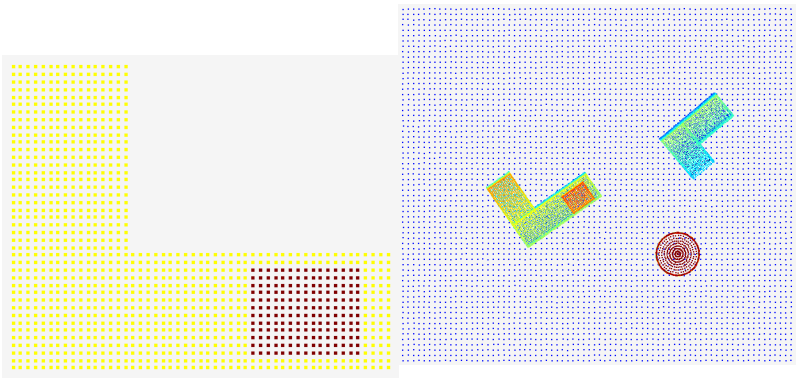
reliably.

For a more systematic treatment of ICP, point cloud registration, and 3D pose estimation, see the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018). The ICP algorithm itself has two classic origins: Besl and McKay gave the point-to-point formulation based on iterating to the closest point (Besl and McKay 1992), while Chen and Medioni, registering multiple range images, proposed the faster-converging point-to-plane error metric (Chen and Medioni 1992) — exactly the 58-vs-9-iteration difference in this chapter's experiments; a systematic comparison of the sampling, matching, and rejection variants is given in the survey by Rusinkiewicz and Levoy (Rusinkiewicz and Levoy 2001). Each round's closed-form optimal rigid transform, obtained by the SVD of the cross-covariance matrix after centroid subtraction (Kabsch/Umeyama), was first rigorously derived in the paper by Arun, Huang, and Blostein (Arun, Huang, and Blostein 1987).

40 3D Matching

ICP (Chapter 39) solves the problem of "pressing two already roughly aligned point clouds into subpixel agreement" — it is a refinement engine, but congenitally near-sighted: let the initial value be off by a few tens of degrees and it locks into the wrong local minimum, never to climb out again. Yet what the camera on a production line sees is never an isolated blob of points but a frame of cluttered scene: the ground, the bin walls, neighboring workpieces, noise — with the part to be found buried inside and its pose entirely unknown. **3D matching** is responsible for precisely this "from nothing to something" step — locating a known model inside a cluttered scene point cloud and giving its rough **6 degrees of freedom (6DoF) pose**, then handing this coarse pose off to ICP for refinement. It is the three-dimensional counterpart of 2D **template matching** (Chapter 16) and **shape matching** (Chapter 17), and the eyes of robotic grasping and assembly: only once the part's "where and which way" is known can the arm even begin to reach for it.

This chapter's model is an asymmetric **L-shaped bracket** of about 5400 surface points: a long arm and a short arm form the L, with a boss added at the tip of the long arm (Figure 40.1a) — the boss deliberately breaks the symmetry the L itself retains, making the principal-axis direction uniquely identifiable; the reason is in Section 40.2. The scene (Figure 40.1b) contains two instances of this bracket, each at a different 6DoF pose: instance 1 clean and complete, instance 2 occluded (about 1/3 of the surface removed); in addition there is a ground plane and a distractor cylinder, with Gaussian noise of $\sigma = 0.1\,\text{mm}$ superimposed over the whole scene. The task is to find both instances and pin down their poses, while not mistaking the cylinder for the bracket.



(a) Model: the asymmetric L-shaped bracket template (height pseudocolor), long arm + short arm + top boss

(b) Scene: blue ground + complete L (left, orange) + occluded L (upper right, a segment missing) + distractor cylinder (red), with $\sigma = 0.1$ mm noise

Figure 40.1: The experimental data of this chapter, rendered in orthographic top view with height pseudocolor. The model has about 5400 points; the scene contains two instances at different poses (one occluded), a ground plane, and a distractor cylinder.

40.1 Six-Degree-of-Freedom Pose Search

2D matching searches over three parameters (x, y, θ) — planar translation plus in-plane rotation; what let template/shape matching in the previous chapters score brute-force cell by cell over an angle × scale grid was precisely that the search space is only three-dimensional and that the image pyramid shrinks it level by level. Move to three dimensions and a rigid-body pose has **6 degrees of freedom**: three translations (t_x, t_y, t_z) plus three rotations (about the three axes). As the dimension rises from 3 to 6, the search space does not double but **explodes exponentially** — discretize each rotation axis into K steps and each translation axis into K steps, and the total number of candidates is K^6. The motivation of Chapter 16 that "brute-force scoring cannot hold, a pyramid is mandatory" is, in 3D, magnified to the point of being unavoidable: even at a coarse 30°/10 mm per axis, $K \approx 12$ already means millions of candidates, and each candidate still has to be scored by nearest neighbors over thousands of points.

The conclusion is the same as in 2D, only more pressing: **no brute force, coarse-to-fine is mandatory**. First use cheap means to circle out the few candidates in the scene that "might be the part," each with a coarse pose (**coarse matching**), then run ICP on only these few candidates to press the pose down to the noise floor (**fine registration**). Coarse matching is responsible for collapsing the K^6 search into a handful of hypotheses, and fine matching for wiping out the few-degree, few-millimeter error of each hypothesis — neither can be spared, and this pipeline is the throughline of the chapter.

The "6" of 6DoF is all the freedom a rigid body has in three-dimensional space: translation 3 + rotation 3. If the part may also scale (the working distance uncalibrated), add 1 more scale degree of freedom — but an industrial 3D camera is already calibrated to millimeters, the scale is usually fixed, and this is one thing 3D matching worries less about than 2D.

40.2 Coarse Matching Methods

The first step of coarse matching is not matching but **scene segmentation**: cutting a frame of point cloud into "individual candidate objects." We reuse the two staples of point-cloud preprocessing (Chapter 38) — first fit the largest plane with **RANSAC** and strip away the ground (the ground is the plane with the most points in the scene; sample three

points at random to fit, count inliers, and take the best over 200 iterations); once the ground is gone, the objects suspended above it separate from one another, and then comes **Euclidean clustering**: voxelize the space and, using a union-find structure, merge points in adjacent voxels whose distance is below a threshold (here 4 mm), so that each connected block is a cluster. After the ground is removed, this chapter's scene splits cleanly into three clusters: the two bracket instances + one cylinder.

Each cluster demands a **coarse pose**. The most naive approach is **principal component analysis (PCA)**: take the cluster's centroid as the translation initial value, take the three eigenvectors of the covariance matrix as the object's three principal axes, and align them with the model's own principal axes to obtain the rotation initial value. The principle is plain — for a slender L-bracket, the point cloud's direction of maximum variance must lie along the long arm, and PCA "sees" this axis at a glance. But PCA has two congenital limitations: first, **the principal axes carry sign and ordering ambiguity** — eigenvectors are arbitrary in sign, and under nearly equal eigenvalues the ordering can swing too, so all 24 right-handed combinations of the three axes are "equally reasonable" and must be tried one by one, disambiguated by "which one makes the model fit the scene best"; second, **sensitivity to occlusion** — delete part of the part and the variance distribution of the point cloud changes, the principal axes skew with it, and the rotation PCA gives can be off by tens or even hundreds of degrees. The latter is exactly the story of instance 2 (Section 40.3).

This chapter generates all 24 PCA candidate poses for each cluster, runs a fast wide-capture-radius ICP on each, and picks the best by **scene coverage** (defined in the next section) — using one cheap "trial-and-error search" to make up for the fragility of PCA's single estimate.

Beyond PCA there is a stronger class of coarse matching: the **point-pair feature (PPF)** — for pairs of points on the model, form a 4-dimensional feature from their distance and the angle between the two normals and vote out the pose; it is more robust to occlusion and clutter and is the cornerstone of Drost et al.'s industrial surface matching. This chapter uses PCA + multi-candidate search to demonstrate the principle; PPF takes considerably more implementation and is left as an extension.

40.3 The Coarse-to-Fine Pipeline

This is the core teaching experiment of the chapter: **coarse matching feeds the initial value, ICP presses down the noise floor**, and the two instances play out both typical scripts of this pipeline in full.

Instance 1 (clean): with the surface complete and the point cloud dense, the PCA principal axes barely skew, and the winning candidate's coarse pose is already at **rotation error** 0.02°**, translation error** $0.01\,$**mm** — under full sampling PCA all but lands it in one step. After ICP takes over it is refined to $0.01^\circ/0.00\,$**mm**, with inlier residual $0.162\,$mm and both scene coverage and model coverage reaching 100%. Here ICP has almost nothing to do, because the coarse pose already sits near the noise floor.

Instance 2 (occluded): after about 1/3 of the surface is removed, the principal axes skew severely, and the best of the 24 candidates' coarse poses is off by as much as **rotation** 100.5°**, translation** $21.8\,$**mm** — taken on this number alone, PCA has all but failed. Yet it still places the model into the correct cluster, at the correct rough position, enough for ICP under a wide capture radius to progressively tighten the nearest neighbors and "twist" the model back bit by bit. In the end ICP converges to $0.38^\circ/0.90\,$**mm**, residual $0.400\,$mm. The reduction across the two stages is striking: **rotation** $100.5^\circ \to 0.38^\circ$ **(about** $\times 266$**), translation** $21.8 \to 0.9\,$**mm (about** $\times 24$**)**.

Figure 40.2 draws the two ICP convergence trajectories together (the vertical axis is residual on a log scale). The orange line is the occluded instance: starting from the roughly $4\,$mm residual of the coarse pose, it steps down with the iterations to about $0.4\,$mm — each step the result of a tightened capture radius taking in more correct correspondences and shedding wrong ones. The blue line is the clean instance: pressed to the bottom from the start and nearly horizontal — it was at the noise floor all along, and ICP only confirms rather than carries. This figure is the closing footnote to Chapter 39's assertion that "ICP needs a good initial value": **a good initial value (clean) leaves**

The clean instance's residual $0.162\,$mm falls right around the noise floor: for points with independent per-axis $\sigma = 0.1\,$mm, the RMS distance to the true surface is about $\sigma\sqrt{3} \approx 0.17\,$mm. Once ICP converges to this order it is at the end of the line — any smaller would be overfitting the noise. The occluded instance's $0.40\,$mm is slightly higher, the limit under partial data rather than an algorithm defect.

ICP nothing to do, a bad initial value (occluded) lets ICP show its worth, and a hopelessly bad initial value makes ICP fail outright — and this chapter, by "multi-candidate + wide radius," rescues instance 2's initial value to within ICP's reach.

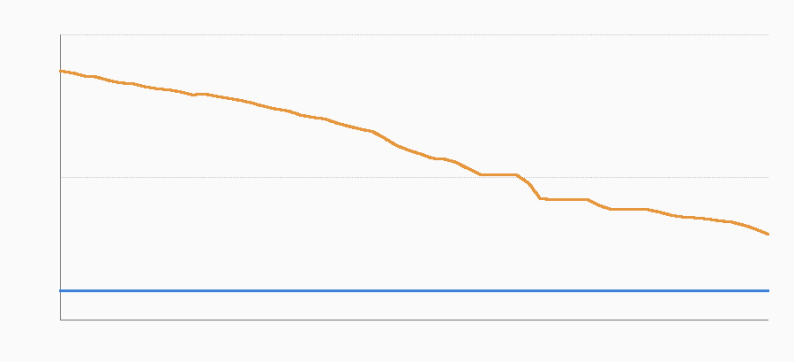


Figure 40.2: ICP convergence trajectories (horizontal axis iteration count, vertical axis inlier residual on log scale, gridlines at 0.1/1/10 mm). Orange = occluded instance, stepping down from about 4 mm to about 0.4 mm; blue = clean instance, hugging the noise floor (about 0.16 mm) almost horizontally throughout.

40.4 Occlusion and Coverage

In Chapter 17, 2D occlusion makes the score degrade by the visible fraction; 3D matching has to split "coverage" into **two oppositely directed coverage measures**, which measure different things.

- **Scene coverage**: after placing the model at the candidate pose, **how many points in the cluster land near the model surface**. It asks "of this clump of points I found, how many can the model explain." Occlusion only deletes scene points, and **the remaining points still lie on the model**, so scene coverage is **insensitive** to occlusion — and for exactly this reason it is chosen as the

acceptance criterion: high coverage means accept this match.

- **Model coverage**: conversely, **how many of the model's points found a corresponding scene point**. The occluded portion of the part has no scene points to support it, so these model points go "unclaimed," and model coverage **drops directly** with occlusion — which makes it just right for measuring **how severe the occlusion is**.

The experiment plays out their division of labor clearly. After occlusion removes about 1/3 of the surface: **model coverage falls from the clean** 100% **to** 46.5% (nearly halved, faithfully reflecting the amount of occlusion), while **scene coverage drops only from** 100% **to** 79.3% (still well above the acceptance threshold). In Figure 40.3 the blue occluded instance is visibly missing a stretch of geometry, yet its remaining points still seat snugly against the model — this is exactly the picture of "high scene coverage, low model coverage." Were model coverage used as the acceptance criterion, the occluded instance would be wrongly rejected; with scene coverage, it passes smoothly.

Figure 40.4 is the final match: the model is overlaid back onto the scene at the two recovered poses, red at the clean instance and magenta at the occluded instance, both contours seating snugly; the cylinder, for want of coverage, is not overlaid. That the occluded instance can be localized successfully is the combined force of "PCA alone is not enough, naive ICP alone is not enough, the two in relay are enough" — which is of a piece with the logic of Chapter 17's 2D shape matching that "the score degrades by the visible fraction, and minScore is the tolerable occlusion fraction," only that 3D splits "one score" into "two coverages."

The distractor cylinder is the counterexample: its geometry has nothing to do with the L-bracket, and however the model is placed, the cluster points never seat against the model surface — **scene coverage is only about** 5.4%, far below the acceptance threshold, and it is cleanly rejected. The coverage criterion must both catch the genuine occluded part and block false targets of the wrong shape, and the cylinder is the touchstone for the latter.

40.5 SciVision Implementation

As usual, first the honest record of what the SDK actually does in testing. SciVision provides two sets of 3D matching interfaces: `SciSv3DSurfaceMatch`

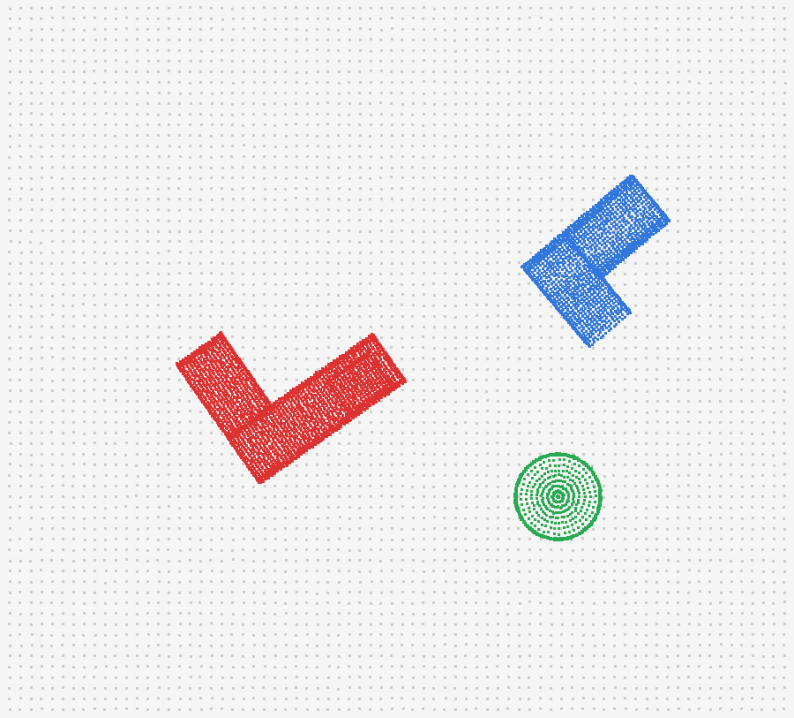


Figure 40.3: Scene segmentation result: gray = ground (already marked out by RANSAC), the three red/blue/green clusters being the Euclidean-clustering output — red the complete L, blue the occluded L (visibly missing a stretch of long-arm geometry), green the distractor cylinder.

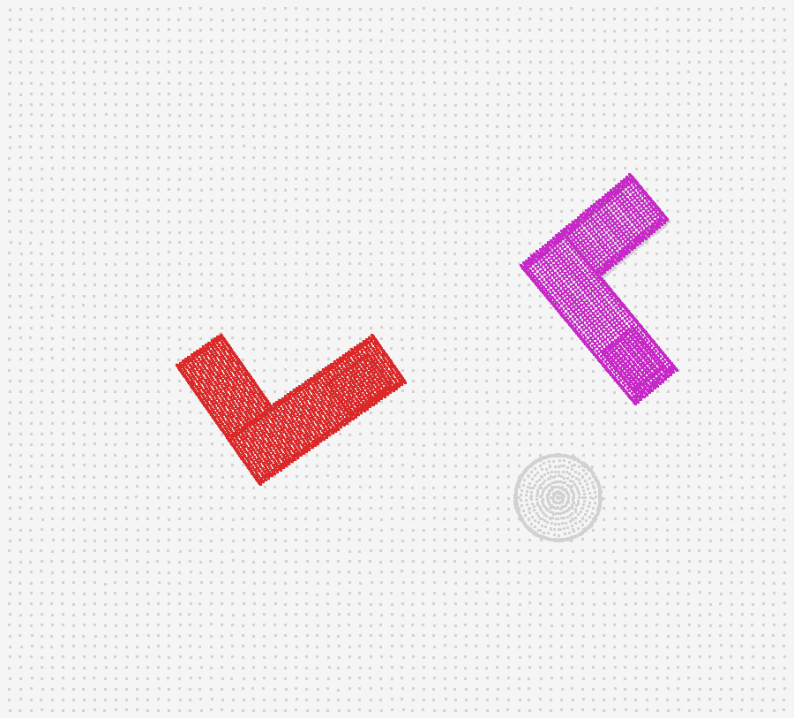


Figure 40.4: Match result: the model is overlaid back onto the gray scene at the two recovered 6DoF poses, the clean instance red and the occluded instance magenta, both fitting; the distractor cylinder, for want of coverage, is not overlaid.

(`CreateSurface3DModel`/`FindSurface3DImageModel`, range-image surface matching) and `SciSv3DPointsMatching` (`GetTransformationICP`/`GetTransformationPCA`, point-cloud registration). This chapter calls each via a subprocess probe, and both **fail inertly**: the return code is `rc=0` (no error, no crash), but surface matching finds **0 matches** and point-cloud ICP/PCA recovers a known translation as $(0, 0, 0)$, while the DLL also prints `"Directory does not exist."` to `stderr` — the same signature as several 3D modules in Part IX of this book (apparently a missing runtime resource directory). The SDK being unusable, the content of this chapter is carried by a **hand-written pipeline**.

The hand-written pipeline has four steps, with the key fragments below.

```
// 1) RANSAC ground removal: 200 random three-point plane fits, take the one with the most inli
std::vector<char> ground = ransacPlane(scene, /*tol*/ 0.4);

// 2) Euclidean clustering: voxel + union-find, tol=4mm, discard clusters with <200 points
std::vector<std::vector<int>> clusters = cluster(rest, 4.0, 200);

// 3) Per cluster: PCA for centroid + principal axes, 24 signed-permutation candidates each ru
//    wide-radius fast ICP, pick the best by scene coverage (disambiguate + rescue skewed axes)
pca(clu, cs, Vs);                        // cluster centroid cs, principal axes Vs
for (auto& P : perms /*24 right-handed permutations*/) {
    matMul(Vs, P.data(), VsP); matMul(VsP, VmT, Rc);   // candidate rotation
    icp(modelScreen, clu, Rt, tt, 25, 14.0, 2.5);      // wide-capture-radius screening
    double sc = sceneCov(model, Rt, tt, cluDown, 0.6, rms);
    if (sc > bestSC) { /* record the best candidate */ }
}

// 4) Run a full ICP from the best candidate as initial value (Kabsch closed form, 3x3 SVD via
best.resid = icp(modelDown, clu, R, t, /*iters*/ 60, /*d0*/ 10.0, /*d1*/ 0.8);
best.sceneScore = sceneCov(model, best.R, best.t, cluDown, 0.5, rmsS);
best.ok = (best.sceneScore > 60.0 && best.resid < 1.5);   // coverage + residual dual criterio
```

Each ICP iteration finds the optimal rigid transform with **Kabsch**: de-mean the paired points, accumulate the covariance matrix C, perform a **Jacobi**

eigendecomposition on $C^{\mathsf{T}}C$ to realize the 3×3 SVD, obtain the rotation from $R = UV^{\mathsf{T}}$ (correcting for reflection so that $\det R = +1$), and find the translation from the difference of the two centroids. The capture radius tightens linearly from d_0 to d_1, which is exactly the source of the step-by-step descent of Figure 40.2's orange line. The whole of the math is fully self-contained — which echoes Chapter 37 and a recurring observation of this book: **point-cloud algorithms (ICP/Kabsch/PCA/PPF/kd-tree) mostly need to be self-developed or to lean on a dedicated library, and on 3D matching the SDK often serves only for IO and cross-validation**. When industrial 3D matching is pushed to any depth, self-development is all but the norm.

40.6 Industry Case

Industry Case: Bin Picking of Randomly Stacked Parts

The same kind of part is randomly stacked in a bin (bin picking), and the arm must find each one's pose and grab them off in turn. The 3D camera scans out a frame of point cloud, and the flow is the same as this chapter's: remove the bin-bottom plane, Euclidean-cluster out the candidates one by one, and on each cluster solve a PCA coarse pose + ICP refinement. The difficulty lies in **mutual occlusion in the stack** — upper parts press on lower ones, most instances have very low model coverage, and accepting by model coverage would miss grasps wholesale. The countermeasure is exactly this chapter's split of criteria: **accept by scene coverage** (occlusion-resistant), then **sort by graspable-face visibility**, grabbing first the one that is most complete and against which the gripper can seat steadily. Interference from the bin walls and neighboring parts is blocked by the coverage threshold — points of the wrong shape never seat against the model, the same way this chapter's cylinder is rejected by its 5.4% coverage. The lesson distills to one line: **the robustness of 3D matching = segmentation quality × coverage criterion**, and the production rule under occlusion is "grab the one you can see most fully first."

40.7 Summary

- **3D matching is the front end to ICP**: ICP (Chapter 39) only refines and needs a good initial value; 3D matching locates a known model inside a cluttered scene and gives a coarse 6DoF pose to feed ICP. It is the three-dimensional counterpart of 2D template/shape matching (Chapter 16, Chapter 17).
- **Six degrees of freedom make brute-force search explode** (K^6), so coarse-to-fine is mandatory: coarse matching (RANSAC ground removal + Euclidean clustering + PCA principal axes/centroid, 24 candidates for disambiguation) collapses the search into a handful of hypotheses, and ICP presses each hypothesis down to the noise floor.
- **The coarse-to-fine relay, measured**: the clean instance is already near-fine from PCA (0.02°/0.01 mm), and ICP brings it to 0.01°/0.00 mm; the occluded instance's coarse pose is off by 100.5°, and a wide-radius ICP rescues it to 0.38°/0.90 mm (rotation ×266, translation ×24). PCA alone and naive ICP alone are both insufficient; only the relay suffices.
- **The division of labor between the two coverages**: scene coverage (fraction of found points, occlusion-resistant) serves as the **acceptance criterion**, model coverage (fraction of the model explained) as the **occlusion severity**. Occlusion drives model coverage 100% → 46.5% and scene coverage only 100% → 79.3%; the distractor cylinder is rejected at 5.4% scene coverage.
- **The honest SDK record**: both `SciSv3DSurfaceMatch` and `SciSv3DPointsMatching` fail inertly (`rc=0`, zero matches / a recovered zero pose), and this chapter is carried by a hand-written pipeline (RANSAC + clustering + PCA + Kabsch ICP) — industrial 3D matching often needs self-development or a dedicated library.

The principles of 3D matching and the engineering practice of point-pair features and surface matching are treated systematically in the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018). Encoding local geometry into a descriptor and then voting for the pose is the common thread

of this family: Drost et al. use point-pair features (PPF) to model globally and match locally, a representative of industrial surface matching (Drost et al. 2010); earlier, Johnson and Hebert recognized 3D objects in cluttered scenes with spin images (Johnson and Hebert 1999); and the Fast Point Feature Histogram (FPFH) proposed by Rusu et al. is one of the most widely used descriptors for point-cloud coarse registration (Rusu, Blodow, and Beetz 2009), offering a more robust source of coarse pose beyond this chapter's PCA.

41 3D Measurement

After a point cloud has been registered and aligned (Chapter 39) and filtered clean (Chapter 37), what the production line actually wants is not a pretty cloud of points but **numbers** — what is this workpiece's step height, is the circular land large enough, is the mating face flat, is there a local raised defect. Computing these numbers and deciding pass or fail on their basis is exactly what 3D measurement does. This chapter is the three-dimensional counterpart of the two-dimensional measurement methodology of Chapter 21: the measurement chain is still **point cloud → fitted primitive → geometric quantity → judgment**, and the role of each stage, how error propagates, and why band-width tolerances and dimensional tolerances differ statistically all map one-to-one onto the 2D case — only the primitives are promoted from lines and circles to planes, and the evaluation switches from pixels to millimeters.

The experimental object (Figure 41.1) is a **real Smart3 range image** (`sample/height01.srt`, 1600×1600, taken from the SDK's "height measurement" example scheme): a precision-machined part, about 19.2×19.2 mm in top view, with a lateral resolution of 0.012 mm/pixel and a vertical 0.001 mm/count. The surface structure is a sunken **grid-channel floor**, above which sits an array of raised **circular boss lands**, each carrying two small dimples, and one of which also has a pronounced local raised **bump** defect. This is real scan data: there are no analytic ground-truth labels, so every number is reported faithfully as a measured value; the roughly 260k unhit pixels (raw count 0) are the invalid background, skipped point by point.

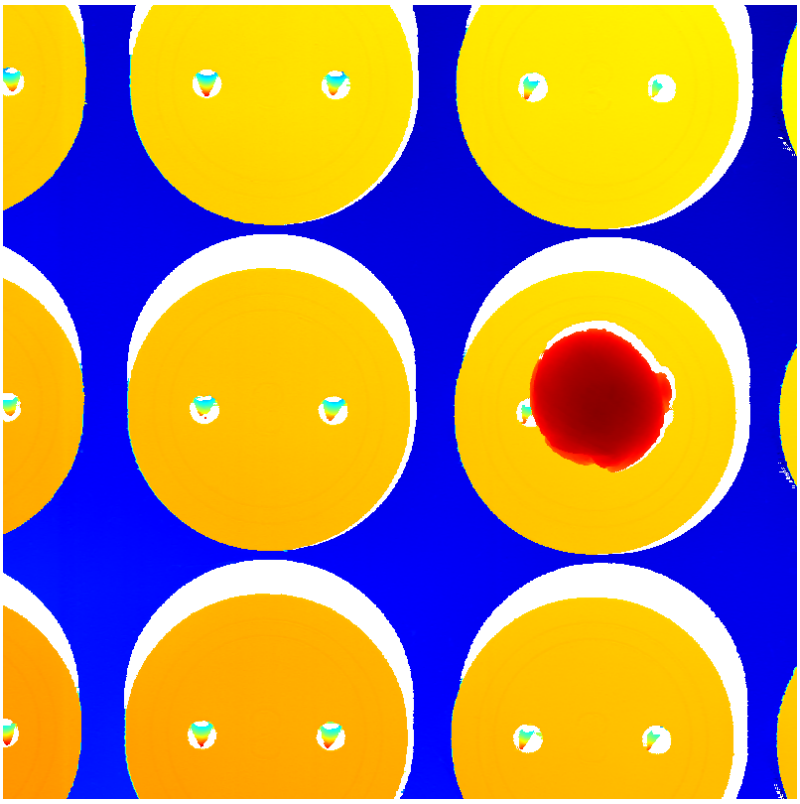


Figure 41.1: Top-view height map of the real range image (jet-encoded, blue 30.5 mm → red 38.2 mm): deep blue is the sunken grid-channel floor, yellow/orange is the array of raised circular lands, the dark-red blob on the right-center land is a raised bump defect, and the two small blue dots on each land are dimples. White is the unhit invalid pixels.

41.1 Fitted Primitives: Planes and Circles

The primitives of 2D measurement are lines and circles (Chapter 14); the workhorse in 3D is the plane. Before fitting, the whole point cloud must first be **segmented** onto its individual feature faces (Figure 41.2). Here the step (4.4 mm) is far larger than the workpiece's fixturing tilt on the stage (± 0.6 mm), so a single global height threshold $z = 33.5$ mm cleanly bisects the channel floor (about 31.2 mm) from the lands (about 35.6 mm); a real production line accomplishes the same step via region growing, normal clustering, or nearest-face assignment to a registered CAD model. After splitting, the **transition walls and flying points must be rejected**: the side-wall pixels between land and channel straddle the dividing threshold, so we fit once, then discard points whose residual exceeds 0.1 mm and refit — what remains is a clean set of plane points.

Least-squares plane fit fits a face's point cloud to $z = ax + by + c$, with the three coefficients solved in one shot from a set of 3×3 normal equations:

$$\begin{bmatrix} \sum x^2 & \sum xy & \sum x \\ \sum xy & \sum y^2 & \sum y \\ \sum x & \sum y & N \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \sum xz \\ \sum yz \\ \sum z \end{bmatrix}.$$

This is precisely the three-dimensional landing of the "least-squares solution of an overdetermined system" from Chapter 2, with the unknowns going from two (a 2D line) to three. Once the coefficients are solved, the plane's unit normal is $\mathbf{n} = (-a, -b, 1)/\sqrt{a^2 + b^2 + 1}$ and the intercept is fixed by the centroid. The fitted channel floor is $z = -0.0151x + 0.0250y + 31.217$, with its normal tilted **1.675°** from vertical — this is the workpiece's actual fixturing tilt on the stage; in real data it is always present, and when measuring the step and parallelism below we must work along the datum-face normal rather than along the z axis, so as not to bake this tilt into the result.

The robustness of plane fitting shares the same origin as in 2D: a few outliers (flying points, side-wall pixels) pull the least-squares datum off, and industrial implementations commonly layer on an iterative reweighting (IRLS) or a RANSAC pre-screen. This chapter uses the most elementary "fit → reject by residual threshold → refit" in two rounds, already enough to squeeze the channel floor's RMS down to **19.0 µm**.

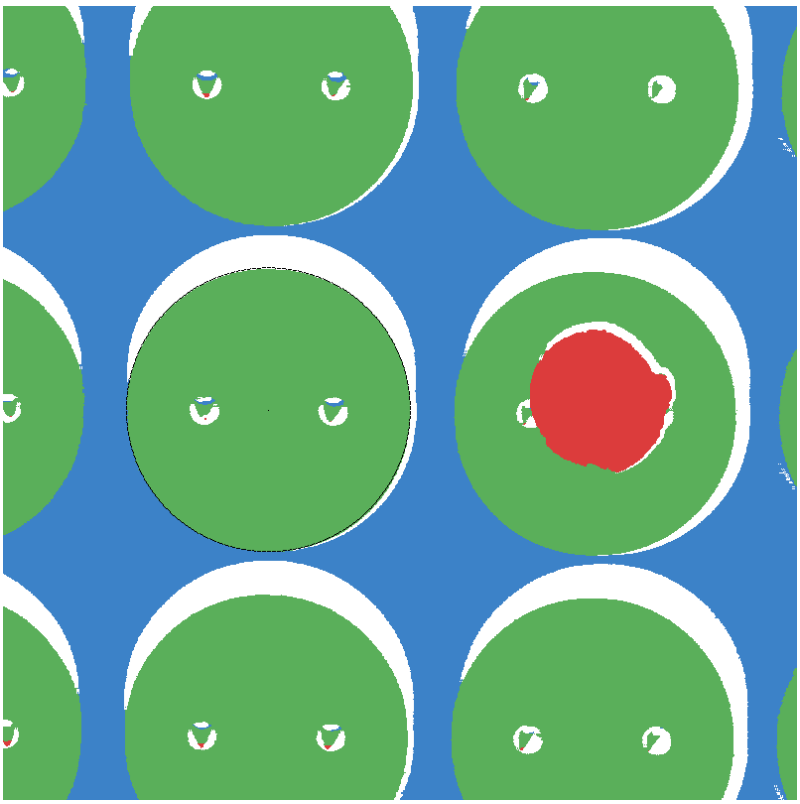


Figure 41.2: Point cloud segmented and colored by height band (top view): blue is the channel floor, green the circular lands, red the bump defect, white the side walls/invalid. The black ring overlaid on the middle-left land is the land outline circle from the Kasa circle fit below.

Circle fit and "what a range image can see". The diameter of a circular land is measured with a **Kasa circle fit**: project the outline points of the land's connected component onto the xy plane and solve a 3×3 normal-equation system isomorphic to the plane fit, yielding the center and radius in one step. There is a key realization about real data here — **a range image is a 2.5D height field that sees only upward-facing surfaces, not vertical side walls**. So a land leaves only a circular outline in the range image (from which a diameter can be fitted), while its **side wall** as a cylinder segment is simply not sampled; fitting a true three-dimensional cylinder (axis + radius, five unknowns) requires multi-view scanning or a full point cloud, which a single top-down range image cannot provide. Here we take a non-edge, largest interior land, bin its boundary by angle and keep only the maximum-radius outline point in each direction (cleanly excluding the boundaries of the two inner dimples), and fit a diameter of **6.8608 mm** (radius 3.4304 mm), with a roundness (radial peak-to-valley) of **86.4 µm** (Figure 41.3).

41.2 Step, Parallelism, and Height Measurement

Once the primitives are in hand, the geometric quantities are closed-form formulas that themselves introduce no new error (the same as in Chapter 21).

Step (plane-to-plane distance) — exactly the core of this sample's "height measurement". The height difference of the land relative to the channel floor equals the signed perpendicular distance from the land's centroid along the channel-floor normal to the floor. Measured at **4.3915 mm**. Each face is fitted from nearly a million points, and the step fully inherits their sub-micron stability. It is worth stressing "along the normal" rather than "along the z axis": the workpiece is fixtured with a 1.675° tilt, and taking the vertical height difference directly would differ by a factor of $\cos\theta$ (about a 4 µm systematic bias), so engineering practice always takes the step as the perpendicular distance along the datum-face normal.

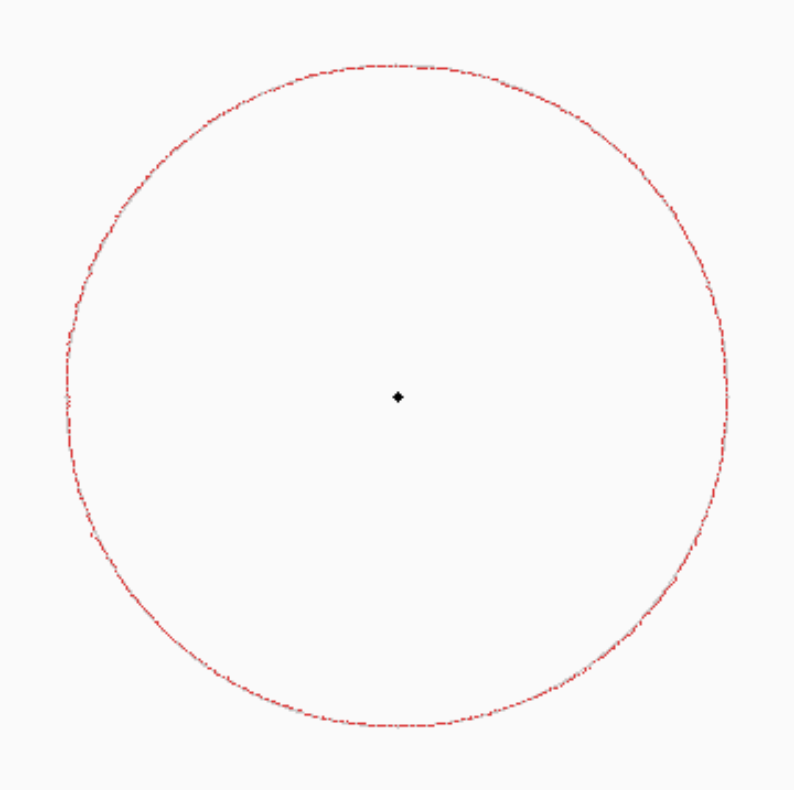


Figure 41.3: The selected circular land's outline points (red) overlaid with the Kasa fitted circle (gray), with the black dot the fitted center. The 720 angularly-sampled outline points hug the fitted circle, with a roundness of just 86.4 µm (RMS 12.0 µm), showing that the land's outer edge really is a true circle.

Face-to-face angle / parallelism. The angle between two planes is given by the dot product of their normals, $\theta = \arccos|\mathbf{n}_1 \cdot \mathbf{n}_2|$ — the 3D counterpart of the line-line angle of Chapter 21 (a normal is to a plane as a direction vector is to a line). The normal angle between the channel floor and the lands measures **0.1085°**, close to 0, confirming that the upper and lower face groups of this machined part are indeed parallel — that is, the 1.675° overall tilt is a fixturing-induced rigid-body pose shared by both faces, which cancels out upon differencing and does not contaminate the step. This "common-mode tilt cancels automatically when differencing between faces" is the fundamental reason 3D measurement evaluates against a datum face rather than the world coordinate system.

Plotting the x–z cross-section of the land array along the row through the bump peak (Figure 41.4), the step and the protrusion are immediately clear: the black polyline drops about 4.4 mm back and forth between the land level (green line) and the channel floor, and the dome bulging above the green line in the center is the bump defect.

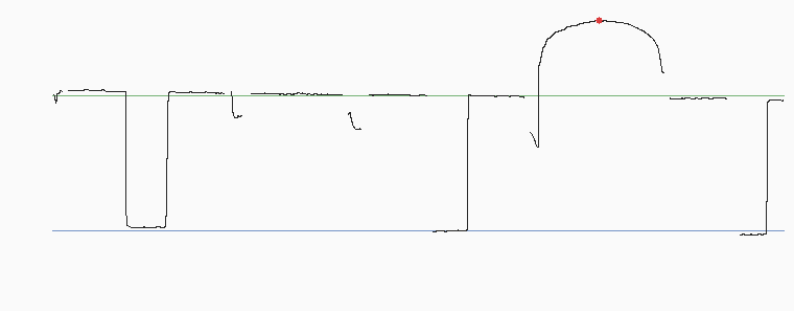


Figure 41.4: The x–z cross-section at the row through the bump peak: the black polyline is the real height along that row, the green horizontal line marks the land datum and the brown line the channel-floor datum; the polyline dropping 4.4 mm back and forth between them is the step, and the dome rising above the green line with a red dot at its apex is the bump defect (2.518 mm above the land).

Pulling the measured geometric quantities together

(Table 41.1). Here two classes of quantity must be distinguished: averaging-class quantities like the step and the diameter become monotonically more stable as the point count grows, with the deviation settling at the $\sigma/\sqrt{N}$ floor; whereas the band-width-class quantities of the next section — flatness, roundness — do exactly the opposite, with the expected range growing as more points are added, so the number is pushed up by the sampling density. The same point cloud, the same noise, yet the two classes respond to "point count" in completely opposite directions — this is the most easily misread point in a 3D measurement report, and is why band-width quantities must always be reported together with the sampling conditions.

Table 41.1: 3D measurement results from the real range image (no ground-truth labels, reported as measured values; flatness/roundness are band-width quantities that vary with sampling density)

Geometric quantity	Measured	Note
Step / channel floor → land (mm)	4.3915	along datum normal, this sample's main measurement
Parallelism floor land (°)	0.1085	0, confirms the two faces parallel
Fixturing tilt (°)	1.675	floor normal vs vertical, rigid-body pose
Circular land diameter (mm)	6.8608	Kasa circle fit of the outline
Land roundness (radial P2V, µm)	86.4	RMS 12.0
Channel floor flatness, least-squares P2V (µm)	225.5	RMS 19.0
Channel floor flatness, minimum zone (µm)	194.3	least squares

Geometric quantity	Measured	Note
Land-array flatness, least-squares P2V (µm)	255.4	RMS 22.3
Land-array flatness, minimum zone (µm)	234.2	least squares
Bump defect above land (mm)	2.5180	lateral scale 3.44 mm

41.3 Flatness, Roundness, and GD&T

Beyond dimensions, drawings also carry the frames of geometric dimensioning and tolerancing (GD&T), which speak not of "how much is measured" but of a **tolerance zone**: the toleranced face must lie entirely within a band of some thickness. This is fully consistent with the 2D tolerance-zone thinking of "parallelism/roundness/straightness" (Chapter 21), only the band is promoted from between two lines in 2D to between two faces in 3D. The zone for **flatness** is the thickness between two parallel planes, within which all points of the toleranced face must lie; the zone for **roundness** is the ring between two concentric circles. The starting point of this language is assembly function: as long as the toleranced face lies wholly within the zone, whatever its specific undulations, its mating with the counterpart is guaranteed. The universal recipe for evaluating them by vision is the same as in 2D: take the set of toleranced points, compute deviations against the evaluation datum, and **max−min is the minimum band width needed to contain the point set**.

Flatness. With the fitted plane as datum, the max−min of the points' signed perpendicular distances is the peak-to-valley value (P2V). The channel floor measures **225.5 µm** (least-squares evaluation, with an RMS of only 19.0 µm), and the land array **255.4 µm** (RMS 22.3 µm) — the land-array number is slightly larger because it evaluates several mutually independent lands together, folding in the

coplanarity error between lands rather than just the machining undulation of a single land. Figure 41.5 draws the residuals of the two faces as diverging heatmaps (±60 µm): the channel floor's (left) long-range red-blue gradient is a large-scale plane waviness, while each land's (right) concentric red-blue rings are a slight central doming — these are real topography left by machining and scanning together, not random noise. Note that flatness is an **extreme-value statistic**: the RMS is only a couple dozen microns, yet the P2V reaches a couple hundred, precisely because it is determined by the two most extreme points across the whole face, and the denser the sampling, the higher the chance of catching an extreme point and the larger the number.

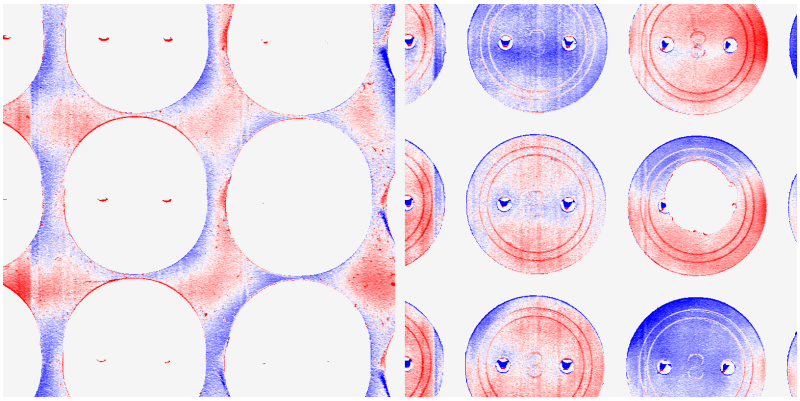


Figure 41.5: Residual heatmaps of the two faces (blue-white-red diverging color bar, half-range ±60 µm). Left: the channel floor, whose long-range red-blue gradient is a large-scale plane waviness; right: the land array, where each land's concentric red-blue rings are a slight central doming. Both show structured topography rather than white noise, indicating the residuals contain real machining and scanning features.

Roundness and the bump defect. The outline roundness of the land from the previous section is **86.4 µm** (Figure 41.3) — the outline points hug the fitted circle evenly, with no systematic ellipse or polygon, showing the land's outer edge really is a true circle. The bump defect, by contrast, uses a different criterion: its peak pixel rises **2.5180 mm** above the

land datum (lateral scale about 3.44 mm), located by connected-component segmentation and taking the perpendicular distance of the highest point in the component along the land normal. Such a local protrusion directly props up the mating face in assembly, and is a defect quantity that must be reported separately — a different language from a global band-width quantity like flatness.

Datum dependence. For the same set of points, switching the evaluation datum changes the flatness number: with the **least-squares** plane as datum the channel floor's P2V is **225.5 µm**, while with the **minimum zone** datum (fine-tuning the plane's tilt to make the containing band thinnest) it is **194.3 µm**; the land array correspondingly gives **255.4 µm** and **234.2 µm**. This is the 3D reprise of the 2D "datum dispute over roundness" of Chapter 21 — the minimum zone gives the thinnest band by definition and is the very embodiment of the GD&T semantics; least squares is the most robust but its band width is systematically larger. So **any reported geometric tolerance must be reported together with its evaluation method**, otherwise the numbers from two machines are not comparable.

Minimum zone ≤ least squares, always. The minimum-zone datum is chosen specifically, among all possible datum planes, as the one giving the thinnest containing band, and the least-squares plane is merely one particular solution among them, so the minimum-zone evaluation value is necessarily no larger than the least-squares evaluation value (in this chapter the channel floor 194.3 ≤ 225.5 and the land array 234.2 ≤ 255.4 — both pairs verify this inequality).

41.4 SciVision Implementation

The real range image is loaded with `Sci3DFileOperation::LoadRangeImage` (link `Sci3DFileOperation.lib`; `rc=0` means success; it prints a harmless "Directory does not exist." to stderr). `GetValue(row,col)` returns the raw count, which times `ResolutionZ()` gives the physical height (mm); a count of 0 is an invalid pixel. The geometric quantities are provided by `SciSv3DGeometryMeasure`, and the geometric tolerances by `SciSv3DGDTTools`. Compared with 2D there is one **key difference** that must be written down faithfully: **the 3D GDTTools return deviation values in real mm units**, rather than the normalized $[0, 1]$ scores of the 2D `SciSvGDTTools` in the neighboring chapter Chapter 42 — this chapter's `ChebyshevFlatness` returns 0.19731 and

`Parallelism` returns 0.26143, both genuine millimeter deviations that can be thresholded directly against the drawing tolerance.

The per-API on-machine measured status (the 3D module is broadly crash-prone/silent, see the engineering conventions for details; every SDK call is fed the same robustly-rejected point set as the hand-written main chain):

- **Working and matching the hand-written version**: `LeastSquarePlaneFit` (channel floor meanErr **0.01905 mm = 19.0 µm**, agreeing to four digits with the hand-written RMS; land 0.02229 mm), `AngleFromPlanes` (**0.10861°**, agreeing to four digits with the hand-written parallelism 0.1085°), `ChebyshevFlatness` (minimum zone **0.19731 mm = 197.3 µm**, differing from the hand-written minimum zone 194.3 µm by only 3 µm, forming a strong cross-validation), `Parallelism` (**0.26143 mm = 261.4 µm**, of the same magnitude as the hand-written land flatness 255.4 µm).
- **Failing**: `PlanesDistance` returns error code 123501017 and yields 0 (a defect in its strict-parallelism tolerance check) — so **the step is computed by hand instead** (the perpendicular distance of the land centroid along the channel-floor normal, 4.3915 mm).
- **Inert, no output**: `ResidualFlatness` returns 0, unusable.

```
SciRangeImage ri;  SCIMV::Sci3DFileOperation fop;
SciVar path("sample\\height01.srt");
long rc = fop.LoadRangeImage(path, &ri, false);
double z_mm = ri.GetValue(row, col) * ri.ResolutionZ();

SCIMV::SciSv3DGeometryMeasure gm;
Sci3DPlane floorPlane, bossPlane;  double meF = 0;
rc = gm.LeastSquarePlaneFit(floorA, 5.0, 5.0, 1, &floorPlane, &meF);
double ang = 0;
rc = gm.AngleFromPlanes(floorPlane, bossPlane, &ang);

SCIMV::SciSv3DGDTTools gdt;
```

On this machine the 3D module is broadly crash-prone or silently returns 0, so SDK calls are uniformly placed in a **subprocess probe**: after the main process has produced its figures and numbers, it re-invokes its own `sdkprobe` branch via `system()` to poke the SDK, and a crash merely exits the subprocess without dragging down the main chain. This phase separation of "hand-written main chain, SDK as corroboration" is the uniform engineering strategy across the 3D chapters of Part IX of this book.

```
double fCheb = 0, par = 0;
rc = gdt.ChebyshevFlatness(floorA, 1, 0.0, &fCheb);      // minimum zone 0.19731 mm   hand-writ
rc = gdt.Parallelism(bossA, floorPlane, 0, 0.0, &par);   // 0.26143 mm (real mm units)
// step: PlanesDistance fails (123501017) -> hand-written land-centroid perpendicular = 4.3915
```

The engineering strategy is consistent with 2D: **the entire measurement main chain uses hand-written least squares** (plane/circle fitting are deterministic, reproducible "mathematical objects"), and the SDK is placed in a subprocess probe for corroboration. This brings two benefits: first, the hand-written implementation is deterministic and reproducible, unaffected by 3D-module crashes, so the main chain can always produce numbers; second, the usable SDK APIs serve as an independent second implementation for cross-validation — `ChebyshevFlatness` independently reproduces the minimum-zone flatness through a completely different code path (197.3 µm vs 194.3 µm), and `LeastSquarePlaneFit`'s meanErr aligns exactly with the hand-written RMS at 19.0 µm, and this "two implementations agreeing independently" is far more trustworthy than the self-corroboration of a single implementation. The complete project (including the generation of the five figures and all the prints of the subprocess probe) is in `code/3d_measurement/`.

Industry Case: Flatness Acceptance of Mating Faces

A machined mating face required flatness at the few-dozen-µm level, yet acceptance was repeatedly mired in disputes. The supplier first accepted with a coordinate measuring machine (CMM) probing points: sparse sampling (a few dozen points) missed the local warp at the mating-face edge, so the reported flatness was optimistically biased and judged passing. The customer switched to a 3D surface scan with full-coverage sampling; the warp was caught, but among the hundreds of thousands of points the scanner's noise floor and fixturing tilt crept in too, and the flatness was pushed up by the extreme-value statistic — just as in this chapter's real sample, where the channel floor's RMS is only 19 µm yet the P2V reaches 225 µm, and the whole face carries a 1.675° fixturing tilt that, if not evaluated along the datum-face normal, adds yet another systematic bias. More thorny still was the

evaluation method: the customer used the minimum-zone method (194 µm), the supplier the least-squares method (225 µm), and the same face yielded two numbers. The eventual resolution lay not in the algorithm but in the protocol — the inspection protocol stated three things in black and white: the evaluation method (minimum zone vs least squares), the sampling density (upper limit on point spacing), and the filter cutoff frequency. The lesson takes one sentence: **a 3D measurement report must carry the triplet — the value, the evaluation method, and the sampling/filtering conditions**; miss one, and the number has no comparability.

41.5 Summary

- **The measurement chain is isomorphic to 2D**: point cloud → fitted primitive (plane/circle) → geometric quantity (step/angle/diameter) → judgment (flatness/roundness/defect); the geometric-quantity stage is exact formulas that introduce no new error, and the accuracy is entirely propagated from single-point noise through $1/\sqrt{N}$ fitting. This chapter uses a real Smart3 range image with no ground-truth labels, so everything is reported faithfully as measured values.
- **A range image is 2.5D and sees only upward-facing surfaces**: a circular land leaves only a top-view outline (from which a diameter of 6.8608 mm can be fitted), while its cylindrical side wall is simply not sampled — a true 3D cylinder fit needs multi-view or a full point cloud, which a single top-down range image cannot provide.
- **Common-mode tilt cancels when differencing between faces**: the workpiece's 1.675° fixturing tilt is shared by the upper and lower faces, and taking the step as the perpendicular distance along the floor normal (4.3915 mm) and the parallelism at 0.1085° both cancel it cleanly — this is the fundamental reason 3D measurement evaluates against a datum face rather than the world frame.

- **Band-width quantities are extreme-value statistics**: flatness (channel floor 225.5 µm / land 255.4 µm, both with RMS of only a couple dozen µm) and roundness 86.4 µm are all sensitive to sampling density, the opposite of "more accurate with more points" dimensional quantities.
- **Datum dependence must be reported together with the evaluation method**: for the same face, least squares gives 225.5 µm vs the minimum zone 194.3 µm (minimum zone least squares, always), and the SDK's `ChebyshevFlatness` reproduces the minimum-zone value through an independent code path (197.3 µm) — reporting a geometric tolerance without the evaluation method is as good as not reporting it.
- **3D SDK units differ from 2D**: the 3D GDTTools return real mm values (not the 2D normalized $[0, 1]$ scores); the usable APIs agree with the hand-written version (plane-fit meanErr 19.0 µm, angle 0.10861°, minimum-zone flatness 197.3 µm), but `PlanesDistance` fails and `ResidualFlatness` is inert, so the main chain still relies on hand-written least squares with the SDK as corroboration.

For a systematic treatment of uncertainty analysis for 3D geometric quantities and minimum-zone evaluation, see further the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018). The geometric tolerances used in this chapter have their symbolic language and tolerance-zone definitions fixed by geometric dimensioning and tolerancing (GD&T) standards: the international system is ISO 1101 (International Organization for Standardization 2017); for the flatness of a single face, ISO 12781 further specifies the vocabulary and parameters (including the minimum-zone and least-squares reference-plane definitions) (International Organization for Standardization 2011b), with the counterpart for cylindricity given in ISO 12180 (International Organization for Standardization 2011a). The "least squares vs minimum zone" datum dispute touched on repeatedly in this chapter is treated specifically in the form-error fitting literature, where Moroni and Petrò compare the principles and costs of various minimum-zone fitting algorithms (Moroni and Petrò 2008).

42 3D Inspection

The inspection part (Part VI, where Chapter 26 lives) hammered the paradigm of defect detection into a single sentence: **a defect is a deviation from "normal"** — first define the one and only normal, then quantify the endlessly varied deviation. The stage there was a 2D grayscale image, where both normal and deviation were written into pixel brightness. This chapter carries the same paradigm into three dimensions: the object under test is no longer a grayscale image but a **range image (height map)** — each pixel stores not brightness but the true height (mm) of the object's surface at that point. Not a word of the paradigm changes, yet the dimensionality does: in 3D a defect acquires, for the first time, **height, volume, and flatness** — quantities that 2D grayscale simply cannot see. An ink blot 25 gray levels darker than its background can only be told apart in 2D through contrast; but a 0.99 mm step or a 7 mm boss is, in 3D, 0.99 mm and 7 mm — no more, no less. Dimension equals physical quantity, and this is the deepest dividend of 3D inspection over 2D, the thread that runs through this entire chapter.

This chapter runs all its experiments on a **real Smart3 range image** (Figure 42.1, file `001.srt` from the Smart3 1.9.2.2 "3D multi-contour inspection" sample recipe): the part under test is a multi-level molded/machined component, with a top-view field of 32 mm×50 mm, a lateral resolution of 0.02 mm/px (1600×2500), and a height resolution `ResolutionZ` of only 0.01 µm/count (extremely fine quantization). It is "multi-contour" in the literal sense — the surface splits into an upper and a lower plateau (a step of about 1 mm), bearing several contour features of wildly different heights: a **tall central boss** (about 7 mm above the reference surface), a **low boss** on the lower plateau, a **recessed racetrack groove** at the top end together with the raised platform it

encloses, and a few **tiny pits** on the tall boss's top face. Figure 42.1 spreads the levels out at a glance with a jet height encoding: deep blue is the lower plateau, light blue the upper plateau, the central boss dyed deep red rises high, the cyan low boss sits at the bottom, and the racetrack contour at the top is clearly visible. Worth noting is that **only 41.3% of the pixels are valid**: the other 58.7% are marked invalid (raw count `INT_MIN`), corresponding to steep walls, deep bores, and laser no-return regions — this is the biggest difference between real 3D data and synthetic data, and every step below must first answer "does this pixel have data."

42.1 Multi-Contour Segmentation: 3D Blob Analysis

Two-dimensional blob analysis (Chapter 23) is a three-step pipeline of "threshold segmentation + connected components + feature measurement": binarize the image, then organize the heaps of white pixels into objects and measure them. The first tool of 3D inspection is almost its word-for-word counterpart, except the basis for segmentation switches from "grayscale over threshold" to "**height deviation** over threshold" — which is exactly the heart of "multi-contour inspection": cut one range image into a set of contour regions of wildly different heights, then measure each one.

Concretely, first estimate a **reference surface** — the median of all valid heights is a robust baseline (measured $h_{\text{ref}} = -1.694$ mm, landing right on the largest-area upper plateau; the median is immune to a handful of tall feature pixels). Then compute the per-pixel deviation $\Delta h(x, y) = h(x, y) - h_{\text{ref}}$; pixels with Δh above $+\tau$ are "raised candidates," those below $-\tau$ are "recessed candidates," and connected-component analysis runs on them **separately** (raised and recessed apart, to keep an adjacent raised region and recessed region from merging into one blob at their boundary). The tolerance is set at $\tau = 0.35$ mm and the minimum area at 1500 px (0.6 mm^2) to filter out debris.

A range image is also called 2.5D data: it stores a single-valued height $z(x, y)$ on a regular x, y grid, sitting between a 2D image (no height) and a full 3D point cloud (Chapter 37, where each point may have arbitrary x, y, z and can express overhangs and closed surfaces). The native output of laser triangulation (Chapter 33), structured light (Chapter 32), and similar devices is mostly exactly a range image — this chapter assumes it has already been produced and focuses on "once you have the range image, how do you judge contours and defects." At steep walls and deep bores the laser gets no return, and the sensor can only mark the pixel invalid (here with `INT_MIN`); the valid-pixel fraction varies wildly with the workpiece relief.

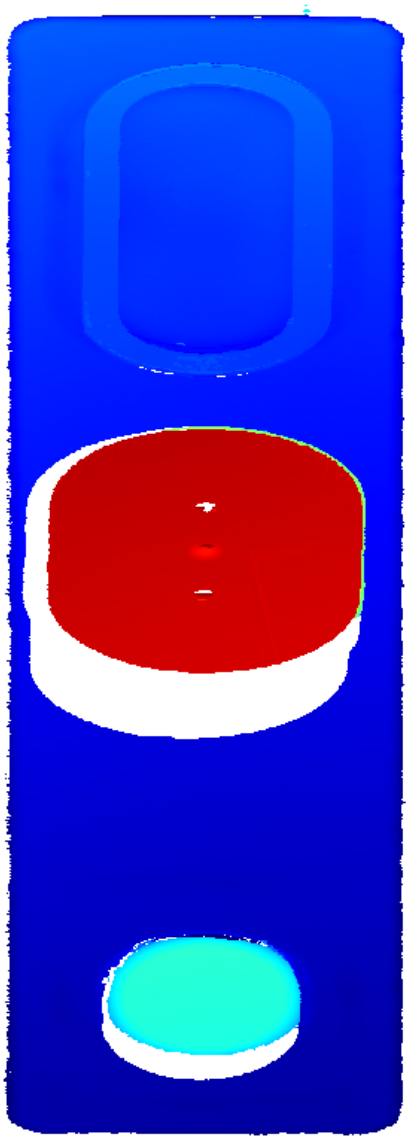


Figure 42.1: Real range image (`001.srt`, 1600×2500, jet height encoding). The part splits into an upper and a lower plateau; the central deep-red feature is a boss about 7 mm above the reference surface, with tiny pits visible on its top face; the cyan feature at the bottom is the low boss; the race-track contour at the top is a raised platform enclosed by a recessed groove. White is invalid pixels (steep walls/no-return, 58.7%). Lateral 0.02 mm/px, field 32×50 mm.

What a 3D contour has beyond a 2D blob is precisely the new criteria that height brings. Besides the 2D suite of area, centroid, and bounding box, each contour also yields: a **signed height deviation** (the sign of the mean distinguishes a raised contour $\Delta h > 0$ from a recessed one $\Delta h < 0$, information a 2D binary image inherently throws away), an **equivalent diameter**, and a **volume** $V = \sum_{(x,y)} \Delta h(x,y) \cdot A_{\text{px}}$ (where $A_{\text{px}} = 0.02 \times 0.02 = 4 \times 10^{-4}$ mm² is the pixel area, the unit is mm³, and it answers directly "how much material does this contour stick out or sink in").

The measurement segments **5 contour features** from this real range image (Table 42.1, Figure 42.2): the most prominent is the **central boss** — equivalent diameter 11.76 mm, 7.15 mm above the reference surface, volume 776 mm³, the geometric protagonist of the whole part; the **low boss** on the lower plateau has an equivalent diameter of 6.23 mm and a height of 2.22 mm (bounding box 7.8×4.8 mm, slightly racetrack-shaped); the **raised platform** enclosed by the racetrack groove at the top spans 54.7 mm² and is only 0.47 mm high; and the whole **lower plateau**, as a large recessed contour (247 mm², mean −0.89 mm), is faithfully separated out — it is the low side of that roughly 1 mm step.

Table 42.1: Multi-contour segmentation results (reference surface $h_{\text{ref}} = -1.694$ mm, $\tau = 0.35$ mm, minimum area 1500 px)

#	Type	Area mm²	Eq. dia mm	Center (x,y) mm	Mean Δh	Peak Δh	Volume mm³
1	Raised	108.6	11.76	(16.3, 24.0)	+7.145	+7.696	+775.9
2	Recessed	247.0	17.74	(16.6, 37.3)	−0.892	−3.868	−220.3
3	Raised	54.7	8.34	(15.9, 5.1)	+0.465	+0.917	+25.5
4	Raised	30.5	6.23	(16.2, 41.9)	+2.222	+2.428	+67.8
5	Raised	2.6	1.81	(16.0, 6.2)	+0.367	+0.406	+1.0

Here the fundamental advantage of 3D inspection emerges: **the criteria are closer to physical quantities**. 2D blobs classify by pixel-geometry quantities such as area and circularity, which shift with resolution and viewpoint; a 3D contour's height (mm), volume (mm^3), and equivalent diameter (mm) are calibrated true dimensions (exactly the "3D metrology gives physical units directly" stressed in Chapter 41). A line such as "boss height must be 7 ± 0.1 mm, step must be 1 ± 0.05 mm" can be written into the process specification verbatim, with no translation into pixels — the unification of dimensionality lets the rulebook and the algorithm speak the same language.

42.2 Cross-Section Measurement

Blob analysis answers "where the contours are and how large their volume," but a production line often needs a sharper question still: along a **specified straight line**, what exactly does the surface height profile look like? This is **profile/cross-section measurement** — extract the height sequence along a line segment on the range image to obtain a one-dimensional profile curve. It is the caliper of the three-dimensional world (Chapter 20): a caliper finds edges and measures distances along a search line on a grayscale image, while a cross-section reads heights and measures steps and peaks along a line on a range image. The two are isomorphic — both reduce 2D/3D data along a line down to 1D, then resolve it with classical one-dimensional signal tools.

The experiment extracts a cross-section along the vertical center line $x = 16.1$ mm, which passes through both the tall and the low boss (Figure 42.3). This line walks through every level in one stroke: the upper plateau, the top groove, the raised platform, the central boss top rising sheer (including the two downward notches of the pits on its top face), the step plunging to the lower plateau, then climbing onto the low boss. The measured profile has **2054 valid points**, with a height range of $[-6.562, 5.987]$ mm: the boss top peaks at **5.987 mm** (at $y = 22.2$ mm), **7.66 mm** above the upper

In practice a volume criterion is often more robust than a height criterion: a sharp tall burr (large height, small volume) and a shallow broad swelling (small height, large volume) usually carry different risks, and looking at peak height alone conflates them. Using height and volume as a **dual criterion** — "NG if height exceeds A or volume exceeds V" — fits the real question of "will this contour cause a failure" better than any single criterion. Here the central boss's 776 mm^3 versus the low boss's 67.8 mm^3 reflects their disparity in bulk far more vividly than their diameters do.

Figure 42.2: Multi-contour segmentation results (height-grayscale base map). Red boxes = raised contours ($\Delta h > 0$: the central boss, the low boss, the raised top platform), blue boxes = recessed contours ($\Delta h < 0$: the lower-plateau step region). The two white dots on the central boss's top face are the tiny pits inspected in the next section.

plateau (−1.668 mm); the upper-to-lower plateau drop is the **step height of 0.988 mm**. All of this is clearly readable in Figure 42.3: two horizontal reference lines mark the two plateaus, the sheer raised platform in the middle is the boss top (red dot marks the peak), the two downward notches on the boss top are precisely the pits inspected next, and the half-height step on the right is the low boss.

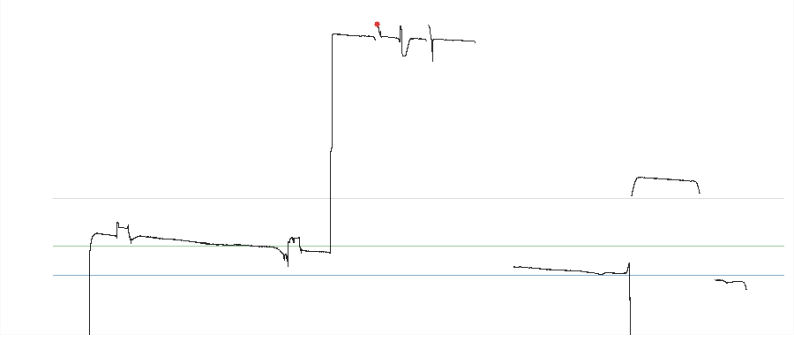


Figure 42.3: Height profile along the vertical center line $x = 16.1$ mm (black curve). From left to right it passes through the upper plateau, the central boss top (peak 5.987 mm, red dot), the step, the lower plateau, and the low boss; the two horizontal lines are the upper/lower plateau references. The two downward notches on the boss top are the top-face pits.

Worth a special note: of this chapter's three SDK 3D inspection modules, the only one **measured working** is cross-section measurement. `SciSv3DProfileMeasure::ExtractData` returns `rc=0`, **2054 sample points**, and a Z range of [−6.562, 5.987] mm along the same line — **point-for-point identical** to the hand-written row-by-row column read — because it is stateless, merely "read along a line + interpolate," and is therefore robust and usable. The value of a cross-section is more than "reading the profile": once 2.5D data is reduced along a line back to 1D, the full suite of one-dimensional subpixel tools of Chapter 14 becomes available — parabolic fitting at a peak yields a subpixel peak position, an edge model measures step width and slope, and a line fit to a

baseline segment estimates tilt. **The cross-section is the bridge connecting three-dimensional data with classical one-dimensional metrology tools.**

42.3 Local Defects and Surface Quality

The previous two sections answered "which contours there are, and how large each is," but inspection has a finer layer still: for a face that is supposed to be flat, **is it itself flat, and does it have local blemishes?** This pulls the scale of inspection from "between contours" into "within a single contour." This section uses the central boss's **top face** as its stage — nominally a flat circular land, it is the purest exercise of the paradigm "define the one and only normal, then quantify the deviation": first fit the top face itself into "normal," then dig out the local dips that deviate from it.

The method has two steps. First, **fit the top face into a reference plane**. Take the boss with a height threshold ($z > 1.5$ mm) and connected components, then erode inward by 14 rings to strip off the slanted walls and edges (leaving only the clean interior top face, 96.85 mm²), and least-squares-fit it to a plane $z = ax + by + c$. The fitted top face carries about a 1.4° clamping tilt $(a, b = -0.0074, -0.0230)$; with the tilt removed, the top-face residual has a **flatness RMS of only 15.4 µm** — on a 7 mm-tall boss top quantized at 0.01 µm, this is a remarkably flat face. Second, **flag the local dips that deviate from the reference plane as defects**: pixels whose residual is below −0.25 mm are "pit candidates," counted by connected components. The measurement detects **2 pits**: one of area 0.440 mm² and depth 644 µm, and one of area only 0.014 mm² and depth 818 µm. Figure 42.4 is a diverging-colormap heat map of the top-face residual (blue = dip, red = bump, white = flush with the reference plane): the whole top face is mostly white (proof that it really is flat), with only the blue-cored red-rimmed patch at the center being the 0.44 mm² pit, and the small dot boxed in black below it being the 0.014 mm²

Where to draw the cross-section line is an engineering decision, not an algorithm problem. A section through the deepest part of a defect gives the maximum deviation, a section along a workpiece edge checks steps and edge collapse, a section perpendicular to a scratch measures groove depth and width. In practice one often first takes a contour centroid from Section 42.1, then automatically lays a section line (or a fan of them) at the centroid for fine measurement — the contour "finds it," the cross-section "measures it precisely," a division of labor identical to the 2D pattern of "blob to locate, then caliper to measure precisely." This chapter's center line is information-dense precisely because it threads every level on a single stroke.

micro-pit; the white void near the top is a laser-no-return deep bore on the top face (invalid pixels, not counted as a pit).

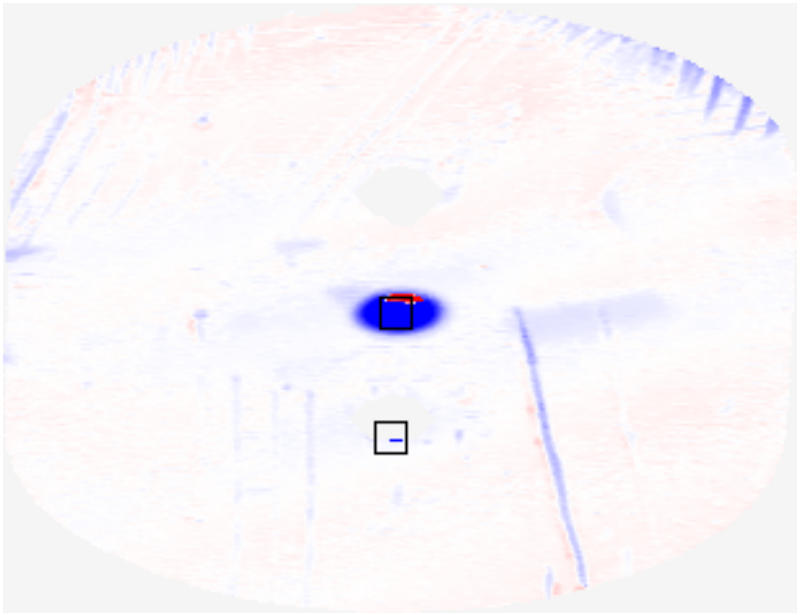


Figure 42.4: Residual heat map of the central boss's top face (clamping tilt removed, diverging colormap ±300 µm; blue = dip, white = flush with the reference plane). Mostly white — flatness RMS only 15.4 µm; the central blue patch is the 0.44 mm²/644 µm pit, the black box marks the 0.014 mm²/818 µm micro-pit; the white void at the top is a no-return deep bore (invalid pixels).

This section pushes the paradigm of defect detection to its logical end: **"normal" need not be the nominal value on a drawing; it can be a reference plane fitted on the spot from the data**. The top-face flatness RMS of 15.4 µm and the pit depths of 644/818 µm are two physical quantities of the same dimensionality, each directly thresholdable against a drawing tolerance — the former answers "is this face flat enough," the latter "is there any damage." One sentence closes this section, and closes the methodology of the entire inspection part: **the definition of a defect determines what quantity, and against what baseline, you use to measure it.**

42.4 Detection Limit and Minimum Detectable Size

The two pits detected in Section 42.3 differ greatly in size: one is 0.440 mm², the other only 0.014 mm² (about 35 pixels, 3–4 pixels wide). The latter already hugs this system's detection limit — it asks not "how deep" but "how small before it can no longer be detected."

First, see the nature of the limit clearly. Both pits are 600–820 µm deep, far above the top face's 15 µm flatness noise floor (SNR about 40), so "whether it is detected" is **unrelated to depth**; the real bottleneck is **lateral size** and the connected-component minimum-area floor. Sweeping the pit-detection minimum area $A_{\min}$ across tight, default, and loose settings (depth threshold fixed at 0.25 mm) gives:

Table 42.2: Minimum detectable pit size (depth threshold fixed at 0.25 mm, sweeping minimum area)

Setting	Min area	Pits detected	Stray specks (1 px)
Tight	6 px	2	**0**
Default	25 px	2	0
Loose	200 px	**1**	1

This table gives a conclusion **strikingly different from synthetic data, and more honest too**. In the 2D world of Chapter 26, the threshold dilemma was "tight = false alarms, loose = missed detections" — tighten the threshold and noise spikes cross the line in patches. But this high-quality real scan's top face is so clean that even pushing the minimum area down to **6 px** raises **no noise specks at all** (false alarms = 0). In other words, **there is no false-alarm-versus-miss dilemma here, only a one-sided size floor**: the loose setting (200 px) drops the 0.014 mm² micro-pit outright (only 1 detection remains), and what truly decides "the smallest detectable" is the 0.02 mm

The SDK has a module named "concentration/density detection" (`SciSv3DConcentrationDetection`), meant to detect **statistical defects** — blemishes whose mean is normal and only the local variance is anomalous (local polishing on a matte surface, orange peel in coatings, porosity in sintered parts), caught by local height variance rather than a height threshold. But on this machine it is **silent with no output** (see Section 42.5), so this chapter demonstrates the same paradigm by the geometric route (residual flatness + pits). Statistical and geometric defects share the same essence — "define normal first, then quantify the deviation" — only the former writes the "deviation" into variance and the latter into height; use the wrong ruler, and even the best algorithm merely computes a zero error precisely.

lateral resolution itself — a 3–4-pixel-wide pit already approaches the boundary where "a single dropped pixel versus a real pit" becomes hard to tell apart. Figure 42.5 puts the two settings side by side: the left (tight, 6 px) boxes both pits, the right (loose, 200 px) keeps only the central large pit, with a blank where the micro-pit sits.

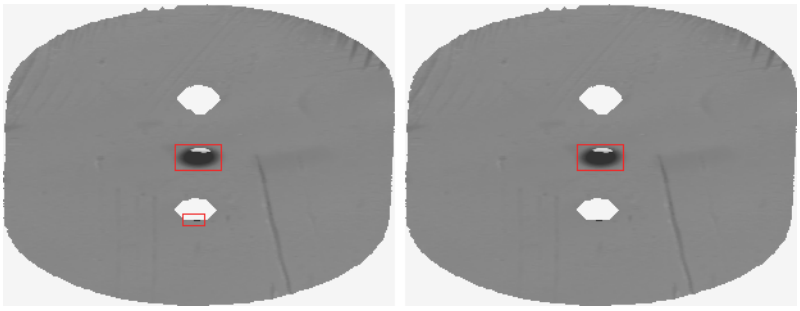


Figure 42.5: Detection limit: comparison of tight/loose minimum-area settings (top-face residual grayscale, red boxes = detected pits). Left: tight 6 px, both pits detected with zero false alarms; right: loose 200 px, the 0.014 mm² micro-pit is missed (only the central large pit remains). The white spot at the top is a no-return deep bore.

The tuning method still descends from the "squeeze from both ends" of Chapter 26: use the **smallest known real defect** to verify detection and get the upper bound (the size floor), and use **good parts** to press down false alarms and get the lower bound — only this clean scan's false alarms are already zero, the lower bound is loose, and the whole bottleneck sits in the upper bound (lateral resolution). To push this size floor lower, what should change is not the area threshold but the acquisition scheme: a lens with higher lateral resolution and a denser scan-line pitch (the sensor selection of Chapter 30 exists precisely to pry this window open). And on this question 3D inspection still holds a card 2D does not: **the depth criterion is in physical units (µm)** — "pit depth must be < 100 µm" is an absolute process quantity reproducible across devices, not drifting with illumination or exposure, which makes 3D threshold tuning steadier and more

portable than a 2D grayscale threshold.

42.5 SciVision Implementation

This chapter records faithfully the measured state of three SciVision 3D inspection modules on this machine, against this **real range image** — itself the book's last practice of its methodology: **every promise of a commercial library must pass a golden experiment.**

The "detection limit" of real data often lies not in the algorithm but in the sampling: the 58.7% invalid pixels (steep walls/deep bores with no return) are the first wall you hit. That no-return deep bore on the top face is the proof — it is not a defect, yet it is a thorough data void that any top-face flatness assessment must exclude first. A 3D inspection engineer's first homework is usually "where is there no data," and only second "is the data that exists judged accurately."

```
// Load the real range image (link Sci3DFileOperation.lib; rc=0 means success;
// it prints a harmless "Directory does not exist." to stderr)
SciRangeImage ri;  SCIMV::Sci3DFileOperation fop;
long rc = fop.LoadRangeImage(SciVar("sample\\001.srt"), &ri, false);
double z = ri.GetValue(r, c) * ri.ResolutionZ();   // raw count × Z-resolution = physical heigh
// invalid pixels are marked INT_MIN(-2147483648) (steep walls/no-return); must be excluded exp

//  Cross-section measurement (measured working! extracts height data along a line)
SciROI line;  line.GenLine(SciPoint(805,120), SciPoint(805,2380));
SCIMV::SciSv3DProfileMeasure pm;  SciFloatArray wz; /* ... */
rc = pm.ExtractData(ri, line, shield, 0, 0, &wxy, &ixy, &wz, NULL, &idxZ, &dist, NULL);
// rc=0, wz.Length()==2054, Z [-6.562,5.987]mm -- point-for-point identical to the hand-written

//  3D blob analysis (measured silent: rc=0 but blobs.Length()==0)
SCIMV::SciSv3DBlobAnalysis ba;  SciRegionArray blobs;  SciMatrix res;
rc = ba.BlobAnalysis3D(ri, roi, base, lo, hi, mn, mx, ft, rt,
                       false, false, &blobs, &res);   // rc=0, blobs=0, res 0×0

//  Concentration/density detection (measured silent: rc=0 but dst is a 0×0 empty image)
SCIMV::Sci3DConcentrationDetection cd;  SciImage dst; /* ... */
rc = cd.DetectConcentration(ri, roi, 0.05, 9,
         SCI_CONCENTRATION_FILTER_MEAN, 4, 4, &diff, &dst, &reg);   // dst 0×0
```

The measured conclusions are clear-cut: `SciSv3DProfileMeasure::ExtractData` is **working** — it returns 2054 sample points along the specified line segment, with a Z range of [−6.562, 5.987] mm, point-for-point identical to the hand-written profile, the only 3D inspection

primitive among the three modules that can be used directly in production. Whereas `SciSv3DBlobAnalysis::BlobAnalysis3D` and `Sci3DConcentrationDetection::DetectConcentration` are both **silent with no output** — the return code is `rc=0` (nominal success), but the former's `blobs` has length 0 and the latter's `dst` is a 0×0 empty image, with no actual output whatsoever. This matches the "silent no-output" signature of several 3D modules in Part IX, so this chapter's main algorithms are all hand-written fallbacks: height-deviation connected-component analysis for multi-contour segmentation (Section 42.1), and least-squares top plane + residual for flatness and pit detection (Section 42.3). To isolate the silent/crash risk, all three SDK modules are invoked through a **subprocess probe** — the main flow first `system()`s a subprocess to run the probe and records its return and output, then proceeds with the hand-written algorithm, so that no SDK anomaly can drag down the main flow. This engineering posture is itself a conclusion: **the SDK maturity of current 3D inspection is markedly below that of 2D**, and most inspection operators still need to be self-developed — no surprise, since 3D inspection is the youngest piece of machine vision, and a standardized operator library is still taking shape.

Industry Case: Online Multi-Contour Inspection of Precision Structural Parts

A connector/structural-part maker runs 100% online 3D inspection on multi-level molded parts like this chapter's, with a laser line-scan sensor. The inspection target is not a single defect but a whole set of **multi-contour dimensions**: boss height (7±0.1 mm), step drop (1±0.05 mm), low-boss diameter, top-face flatness, and whether the top face has pits or foreign matter. Each item maps onto this chapter's three tools — height-deviation connected components separate each contour and measure height/diameter/volume (Section 42.1), the center cross-section measures the step and peak (Section 42.2), and the top-face residual flatness checks local blemishes (Section 42.3). The real engineering difficulties are two. First, **invalid pixels** — at steep walls and deep bores the laser gets no return, 58% of pixels have no data, and every

Why is the cross-section working while the blob and concentration are silent? A reasonable guess: the cross-section is merely "read along a line + interpolate," stateless and with no external resource dependency; whereas the blob and concentration involve internal pipelines of segmentation and filtering, sharing the "Directory does not exist." resource-missing signature with several 3D modules on this machine (see the Part IX measurement records throughout the book). Whatever the root cause, the engineering response is the same — measurement is the arbiter, and silent means self-develop.

assessment must first build a "validity mask," or the voids get mistaken for deep dips. Second, **commercial 3D operators are uneven** — as measured in this chapter, the cross-section works while blob and concentration are silent, and the project schedule must budget for "verify every SDK operator by measurement, one by one; self-develop the unusable ones." The lesson: **the first principle of a 3D inspection project is to ask first "where is there data," then talk about "is the judgment accurate"** — a lesson barely worth worrying about in 2D inspection, but unavoidable once you reach 3D.

42.6 Summary

- **3D inspection follows the "defect = deviation from normal" paradigm (Chapter 26), but with dimensionality upgraded to physical quantities**: in 3D a contour acquires height (mm), volume (mm³), and flatness (µm) — dimensions invisible to 2D grayscale — and for the first time the rulebook and the algorithm speak the same language.
- **Multi-contour segmentation = height-deviation threshold + connected components + physical features** (the 3D counterpart of Chapter 23): 5 contours are separated from the real range image, with the central boss (Ø11.76 mm / 7.15 mm high / 776 mm³ volume), the low boss (Ø6.23 mm / 2.22 mm high), and the 0.988 mm step each measured out; raised and recessed are run as separate connected components to avoid spurious merging.
- **Cross-section measurement is the caliper of 3D** (Chapter 20): reduce 2.5D down to 1D along a center line that threads every level; `SciSv3DProfileMeasure::ExtractData` is measured working, its 2054 points point-for-point identical to the hand-written version, the boss top peaking at 5.987 mm.
- **Local defects fit "normal" into a reference plane on the spot**: the central boss's top-face least-squares flatness RMS is only 15.4 µm, and the residual digs out

two pits (0.440 mm²/644 µm, 0.014 mm²/818 µm) — the definition of a defect determines what quantity, and against what baseline, you use to measure it.

- **In a real high-quality scan the detection limit is a one-sided size floor**: the pits, 600+ µm deep, are far above the 15 µm noise floor, with no false-alarm dilemma; the 0.014 mm² micro-pit (3–4 pixels wide) approaches the 0.02 mm lateral resolution — while the 58.7% invalid pixels (no-return) are the first wall a 3D inspection hits.

For a more systematic treatment of industrial 3D surface inspection (including defect taxonomy and texture and statistical models), read further in the book by Steger et al. (Steger, Ulrich, and Wiedemann 2018). This chapter's "defect = deviation from normal" automated optical inspection paradigm, together with the range-image/2.5D inspection route, is given a full historical and methodological survey in Newman and Jain's review of automated visual inspection (Newman and Jain 1995).

And here, at this point, the book's forty-two chapters close their last page. We set out from the digital image of Chapter 1 — light sampled by a sensor into a grid of gray levels (Chapter 1) — and arrive at this final chapter's 3D inspection, where an algorithm judges, on physical height, whether a 7 mm boss, a 1 mm step, and a 644 µm pit are acceptable. Spread out in between is machine vision's complete engineering chain: imaging sets the ceiling on information, and the contrast that light could not bring out, the steep wall the laser could not reach, no algorithm however strong can conjure; preprocessing paves the way for every step that follows; localization, measurement, inspection, and recognition each do their part, translating pixels layer by layer into "what it is, where it is, how big it is, whether it passes"; and from 2D to 3D adds yet another dimension to all of this, giving contours a height and a volume, and measurement a real millimeter.

But if only one sentence were to be kept from these forty-two chapters, it would not be any specific algorithm. Otsu will be replaced by a better thresholding method, the hand-written connected components by a faster implementation, today's silent SDK perhaps fixed tomorrow. What truly runs through the whole book, and is worth taking with you, is a set of **engineering methodology**: **understand the physics** — know how light, lens, and sensor decide what you can see, including those 58.7% of pixels that simply have no return; **quantify the error** — every number deserves an error bar, and a 15 µm flatness is more honest than the two words "pretty flat"; **face the failure boundary honestly** — the no-return data voids, the blind zone across a steep wall, the silent commercial library: writing them down as they are is a hundred times more useful than covering them up; and **verify everything by experiment** — including, and especially, verifying the commercial library you depend on, because the documentation is a promise, and only a controlled experiment against known behavior (cross-section working, blob and concentration silent) is the truth. This methodology belongs to neither 2D nor 3D, to no single SDK or vendor; it belongs to every engineer who must make algorithms render trustworthy judgments, day after day, on a real production line.

Algorithms will be updated; methodology endures. May you carry it, and go see a wider world.

References

American Society of Mechanical Engineers. 2018. *ASME Y14.5-2018 Dimensioning and Tolerancing*. New York, NY: American Society of Mechanical Engineers (ASME).

Arun, K. S., Thomas S. Huang, and Steven D. Blostein. 1987. "Least-Squares Fitting of Two 3-d Point Sets." *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-9 (5): 698–700.

Automotive Industry Action Group. 2010. *Measurement Systems Analysis (MSA) Reference Manual*. 4th ed. Southfield, MI: Automotive Industry Action Group (AIAG).

Ballard, Dana H. 1981. "Generalizing the Hough Transform to Detect Arbitrary Shapes." *Pattern Recognition* 13 (2): 111–22.

Belongie, Serge, Jitendra Malik, and Jan Puzicha. 2002. "Shape Matching and Object Recognition Using Shape Contexts." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24 (4): 509–22.

Bentley, Jon Louis. 1975. "Multidimensional Binary Search Trees Used for Associative Searching." *Communications of the ACM* 18 (9): 509–17.

Besl, Paul J., and Neil D. McKay. 1992. "A Method for Registration of 3-d Shapes." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14 (2): 239–56.

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning*. Springer.

Blum, Harry. 1967. "A Transformation for Extracting New Descriptors of Shape." In *Models for the Perception of Speech and Visual Form*, edited by Weiant Wathen-Dunn, 362–80. Cambridge, MA: MIT Press.

Borgefors, Gunilla. 1988. "Hierarchical Chamfer Matching: A Parametric Edge Matching Algorithm." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 10 (6): 849–65.

Brown, Duane C. 1971. "Close-Range Camera Calibration." *Photogrammetric Engineering* 37 (8): 855–66.

Brown, Lisa Gottesfeld. 1992. "A Survey of Image Registration Techniques." *ACM Computing Surveys* 24 (4): 325–76.

Canny, John. 1986. "A Computational Approach to Edge Detection." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8 (6): 679–98.

Chen, Yang, and Gérard Medioni. 1992. "Object Modelling by Registration of Multiple Range Images." *Image and Vision Computing* 10 (3): 145–55.

Cooley, James W., and John W. Tukey. 1965. "An Algorithm for the Machine Calculation of Complex Fourier Series." *Mathematics of Computation* 19 (90): 297–301.

Cortes, Corinna, and Vladimir Vapnik. 1995. "Support-Vector Networks." *Machine Learning* 20 (3): 273–97.

Cover, Thomas M., and Peter E. Hart. 1967. "Nearest Neighbor Pattern Classification." *IEEE Transactions on Information Theory* 13 (1): 21–27.

Debevec, Paul E., and Jitendra Malik. 1997. "Recovering High Dynamic Range Radiance Maps from Photographs." In *Proc. 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 369–78.

Douglas, David H., and Thomas K. Peucker. 1973. "Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or Its Caricature." *Cartographica: The International Journal for Geographic Information and Geovisualization* 10 (2): 112–22.

Drost, Bertram, Markus Ulrich, Nassir Navab, and Slobodan Ilic. 2010. "Model Globally, Match Locally: Efficient and Robust 3D Object Recognition." In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 998–1005.

Duda, Richard O., and Peter E. Hart. 1972. "Use of the Hough Transformation to Detect Lines and Curves in Pictures." *Communications of the ACM* 15 (1): 11–15.

Duda, Richard O., Peter E. Hart, and David G. Stork. 2001. *Pattern Classification*. 2nd ed. Wiley.

Fischler, Martin A., and Robert C. Bolles. 1981. "Random Sample Consensus: A Paradigm for Model Fitting." *Communications of the ACM* 24 (6).

Fitzgibbon, Andrew W., Maurizio Pilu, and Robert B. Fisher.

1999. “Direct Least Square Fitting of Ellipses.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 21 (5): 476–80.

Freeman, Herbert. 1974. “Computer Processing of Line-Drawing Images.” *ACM Computing Surveys* 6 (1): 57–97.

Geng, Jason. 2011. “Structured-Light 3D Surface Imaging: A Tutorial.” *Advances in Optics and Photonics* 3 (2): 128–60.

Golub, Gene H., and Charles F. Van Loan. 2013. *Matrix Computations*. 4th ed. Johns Hopkins University Press.

Gonzalez, Rafael C., and Richard E. Woods. 2018. *Digital Image Processing*. 4th ed. Pearson.

Gorthi, Sai Siva, and Pramod Rastogi. 2010. “Fringe Projection Techniques: Whither We Are?” *Optics and Lasers in Engineering* 48 (2): 133–40.

Haralick, Robert M. 1984. “Digital Step Edges from Zero Crossing of Second Directional Derivatives.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-6 (1): 58–68.

Haralick, Robert M., K. Shanmugam, and Its’hak Dinstein. 1973. “Textural Features for Image Classification.” *IEEE Transactions on Systems, Man, and Cybernetics* SMC-3 (6): 610–21.

Haralick, Robert M., Stanley R. Sternberg, and Xinhua Zhuang. 1987. “Image Analysis Using Mathematical Morphology.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 9 (4): 532–50.

Harris, Chris, and Mike Stephens. 1988. “A Combined Corner and Edge Detector.” In *Proc. Alvey Vision Conference*, 147–51.

Hartley, Richard, and Andrew Zisserman. 2004. *Multiple View Geometry in Computer Vision*. 2nd ed. Cambridge University Press.

Hirschmüller, Heiko. 2008. “Stereo Processing by Semiglobal Matching and Mutual Information.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30 (2): 328–41.

Hu, Ming-Kuei. 1962. “Visual Pattern Recognition by Moment Invariants.” *IRE Transactions on Information Theory* 8 (2): 179–87.

Huang, Lei, Mourad Idir, Chao Zuo, and Anand Asundi. 2018. “Review of Phase Measuring Deflectometry.” *Optics and Lasers in Engineering* 107: 247–57.

Huber, Peter J. 1964. "Robust Estimation of a Location Parameter." *Annals of Mathematical Statistics* 35: 73–101.

Illingworth, John, and Josef Kittler. 1988. "A Survey of the Hough Transform." *Computer Vision, Graphics, and Image Processing* 44 (1): 87–116.

International Organization for Standardization. 2006. *ISO/IEC 16022:2006 Information Technology — Automatic Identification and Data Capture Techniques — Data Matrix Bar Code Symbology Specification*. Geneva, Switzerland: International Organization for Standardization / International Electrotechnical Commission.

———. 2011a. *ISO 12180-1:2011 Geometrical Product Specifications (GPS) — Cylindricity — Part 1: Vocabulary and Parameters of Cylindrical Form*. Geneva, Switzerland: International Organization for Standardization.

———. 2011b. *ISO 12781-1:2011 Geometrical Product Specifications (GPS) — Flatness — Part 1: Vocabulary and Parameters of Flatness*. Geneva, Switzerland: International Organization for Standardization.

———. 2011c. *ISO/IEC 15415:2011 Information Technology — Automatic Identification and Data Capture Techniques — Bar Code Symbol Print Quality Test Specification — Two-Dimensional Symbols*. Geneva, Switzerland: International Organization for Standardization / International Electrotechnical Commission.

———. 2015. *ISO/IEC 18004:2015 Information Technology — Automatic Identification and Data Capture Techniques — QR Code Bar Code Symbology Specification*. Geneva, Switzerland: International Organization for Standardization / International Electrotechnical Commission.

———. 2016. *ISO/IEC 15416:2016 Automatic Identification and Data Capture Techniques — Bar Code Print Quality Test Specification — Linear Symbols*. Geneva, Switzerland: International Organization for Standardization / International Electrotechnical Commission.

———. 2017. *ISO 1101:2017 Geometrical Product Specifications (GPS) — Geometrical Tolerancing — Tolerances of Form, Orientation, Location and Run-Out*. Geneva, Switzerland: International Organization for Standardization.

Johnson, Andrew E., and Martial Hebert. 1999. "Using Spin

Images for Efficient Object Recognition in Cluttered 3D Scenes." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 21 (5): 433–49.

Keys, Robert G. 1981. "Cubic Convolution Interpolation for Digital Image Processing." *IEEE Transactions on Acoustics, Speech, and Signal Processing* 29 (6): 1153–60.

Knauer, Markus C., Jürgen Kaminski, and Gerd Häusler. 2004. "Phase Measuring Deflectometry: A New Approach to Measure Specular Free-Form Surfaces." In *Optical Metrology in Production Engineering, Proc. SPIE*, 5457:366–76.

Lewis, J. P. 1995. "Fast Normalized Cross-Correlation." In *Vision Interface*, 120–23.

Lowe, David G. 2004. "Distinctive Image Features from Scale-Invariant Keypoints." *International Journal of Computer Vision* 60 (2): 91–110.

Marr, David, and Ellen Hildreth. 1980. "Theory of Edge Detection." *Proceedings of the Royal Society of London. Series B, Biological Sciences* 207 (1167): 187–217.

Mitsunaga, Tomoo, and Shree K. Nayar. 1999. "Radiometric Self Calibration." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1374–80.

Mori, Shunji, Ching Y. Suen, and Kazuhiko Yamamoto. 1992. "Historical Review of OCR Research and Development." *Proceedings of the IEEE* 80 (7): 1029–58.

Moroni, Giovanni, and Stefano Petrò. 2008. "Geometric Tolerance Evaluation: A Discussion on Minimum Zone Fitting Algorithms." *Precision Engineering* 32 (3): 232–37.

Nayar, Shree K., and Yasuo Nakagawa. 1994. "Shape from Focus." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16 (8): 824–31.

Newman, Timothy S., and Anil K. Jain. 1995. "A Survey of Automated Visual Inspection." *Computer Vision and Image Understanding* 61 (2): 231–62.

Otsu, Nobuyuki. 1979. "A Threshold Selection Method from Gray-Level Histograms." *IEEE Transactions on Systems, Man, and Cybernetics* 9 (1): 62–66.

Perona, Pietro, and Jitendra Malik. 1990. "Scale-Space and Edge Detection Using Anisotropic Diffusion." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12 (7): 629–39.

Pertuz, Said, Domenec Puig, and Miguel Angel Garcia. 2013. "Analysis of Focus Measure Operators for Shape-from-Focus." *Pattern Recognition* 46 (5): 1415–32.

Pizer, Stephen M., E. Philip Amburn, John D. Austin, Robert Cromartie, Ari Geselowitz, Trey Greer, Bart ter Haar Romeny, John B. Zimmerman, and Karel Zuiderveld. 1987. "Adaptive Histogram Equalization and Its Variations." *Computer Vision, Graphics, and Image Processing* 39 (3): 355–68.

Plamondon, Réjean, and Sargur N. Srihari. 2000. "On-Line and Off-Line Handwriting Recognition: A Comprehensive Survey." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22 (1): 63–84.

Reinhard, Erik, Wolfgang Heidrich, Paul Debevec, Sumanta Pattanaik, Greg Ward, and Karol Myszkowski. 2010. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. 2nd ed. Morgan Kaufmann.

Rosenfeld, Azriel, and John L. Pfaltz. 1966. "Sequential Operations in Digital Picture Processing." *Journal of the ACM* 13 (4): 471–94.

Rublee, Ethan, Vincent Rabaud, Kurt Konolige, and Gary Bradski. 2011. "ORB: An Efficient Alternative to SIFT or SURF." In *Proc. IEEE International Conference on Computer Vision (ICCV)*, 2564–71.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. 1986. "Learning Representations by Back-Propagating Errors." *Nature* 323 (6088): 533–36.

Rusinkiewicz, Szymon, and Marc Levoy. 2001. "Efficient Variants of the ICP Algorithm." In *Proceedings of the Third International Conference on 3-d Digital Imaging and Modeling (3DIM)*, 145–52. Quebec City, Canada.

Rusu, Radu Bogdan, Nico Blodow, and Michael Beetz. 2009. "Fast Point Feature Histograms (FPFH) for 3D Registration." In *IEEE International Conference on Robotics and Automation (ICRA)*, 3212–17. Kobe, Japan.

Rusu, Radu Bogdan, and Steve Cousins. 2011. "3D Is Here: Point Cloud Library (PCL)." In *IEEE International Conference on Robotics and Automation (ICRA)*, 1–4. Shanghai, China.

Rusu, Radu Bogdan, Zoltan Csaba Marton, Nico Blodow, Mihai Dolha, and Michael Beetz. 2008. "Towards 3D Point

Cloud Based Object Maps for Household Environments." *Robotics and Autonomous Systems* 56 (11): 927–41.

Salvi, Joaquim, Sergio Fernandez, Tomislav Pribanic, and Xavier Llado. 2010. "A State of the Art in Structured Light Patterns for Surface Profilometry." *Pattern Recognition* 43 (8): 2666–80.

Scharstein, Daniel, and Richard Szeliski. 2002. "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms." *International Journal of Computer Vision* 47 (1): 7–42.

Serra, Jean. 1982. *Image Analysis and Mathematical Morphology*. Academic Press.

Sezgin, Mehmet, and Bülent Sankur. 2004. "Survey over Image Thresholding Techniques and Quantitative Performance Evaluation." *Journal of Electronic Imaging* 13 (1): 146–65.

Shannon, Claude E. 1949. "Communication in the Presence of Noise." *Proceedings of the IRE* 37 (1): 10–21.

Sharma, Gaurav, Wencheng Wu, and Edul N. Dalal. 2005. "The CIEDE2000 Color-Difference Formula: Implementation Notes, Supplementary Test Data, and Mathematical Observations." *Color Research and Application* 30 (1): 21–30.

Shi, Jianbo, and Carlo Tomasi. 1994. "Good Features to Track." In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, 593–600.

Soille, Pierre. 2004. *Morphological Image Analysis: Principles and Applications*. 2nd ed. Springer.

Steger, Carsten. 1998. "An Unbiased Detector of Curvilinear Structures." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20 (2): 113–25.

Steger, Carsten, Markus Ulrich, and Christian Wiedemann. 2018. *Machine Vision Algorithms and Applications*. 2nd ed. Wiley-VCH.

Suzuki, Satoshi, and Keiichi Abe. 1985. "Topological Structural Analysis of Digitized Binary Images by Border Following." *Computer Vision, Graphics, and Image Processing* 30 (1): 32–46.

Szeliski, Richard. 2022. *Computer Vision: Algorithms and Applications*. 2nd ed. Springer.

Taubin, Gabriel. 1991. "Estimation of Planar Curves, Surfaces,

and Nonplanar Space Curves Defined by Implicit Equations with Applications to Edge and Range Image Segmentation." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 13 (11): 1115–38.

Tomasi, Carlo, and Roberto Manduchi. 1998. "Bilateral Filtering for Gray and Color Images." In *Proc. IEEE International Conference on Computer Vision (ICCV)*, 839–46.

Trier, Øivind Due, Anil K. Jain, and Torfinn Taxt. 1996. "Feature Extraction Methods for Character Recognition — a Survey." *Pattern Recognition* 29 (4): 641–62.

Tsai, Du-Ming, and C.-Y. Hsieh. 1999. "Automated Surface Inspection for Directional Textures." *Image and Vision Computing* 18 (1): 49–62.

Tsai, Roger Y. 1987. "A Versatile Camera Calibration Technique for High-Accuracy 3D Machine Vision Metrology Using Off-the-Shelf TV Cameras and Lenses." *IEEE Journal of Robotics and Automation* 3 (4): 323–44.

Tsai, Roger Y., and Reimar K. Lenz. 1989. "A New Technique for Fully Autonomous and Efficient 3D Robotics Hand/Eye Calibration." *IEEE Transactions on Robotics and Automation* 5 (3): 345–58.

Usamentiaga, Rubén, Julio Molleda, and Daniel F. García. 2012. "Fast and Robust Laser Stripe Extraction for 3D Reconstruction in Industrial Environments." *Machine Vision and Applications* 23 (1): 179–96.

Woodham, Robert J. 1980. "Photometric Method for Determining Surface Orientation from Multiple Images." *Optical Engineering* 19 (1): 139–44.

Wu, Lun, Arvind Ganesh, Boxin Shi, Yasuyuki Matsushita, Yongtian Wang, and Yi Ma. 2011. "Robust Photometric Stereo via Low-Rank Matrix Completion and Recovery." In *Computer Vision – ACCV 2010*, 6494:703–17. Lecture Notes in Computer Science. Springer.

Wyszecki, Günther, and W. S. Stiles. 2000. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. 2nd ed. New York: Wiley.

Xie, Xianghua. 2008. "A Review of Recent Advances in Surface Defect Detection Using Texture Analysis Techniques." *ELCVIA: Electronic Letters on Computer Vision and Image Analysis* 7 (3): 1–22.

Zhang, Zhengyou. 2000. "A Flexible New Technique for Cam-

era Calibration." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22 (11): 1330–34.