Lecture 07 — Feature Points

Lecture 7

Kaveh Fathian
Computer Science Department, Colorado School of Mines

Learning Objectives

By the end of this lecture, you should be able to:

- Explain the roles of **detection, description, and matching** in finding image correspondences.
- Interpret the second moment matrix and its eigenvalues, and use the **Harris score** to detect corners.
- Distinguish invariance from covariance and explain why feature detection needs **scale selection**.
- Describe how **SIFT** detects keypoints and constructs descriptors using local gradients, scale, and orientation.
- Compare descriptor distances and apply **nearest-neighbor matching, Lowe's ratio test, and bidirectional matching**.
- Explain the key ideas behind **SURF, ORB, and shape context**, and use OpenCV to extract and match features.

How Can We Overlay These Images to Create a Panorama?

Image 1



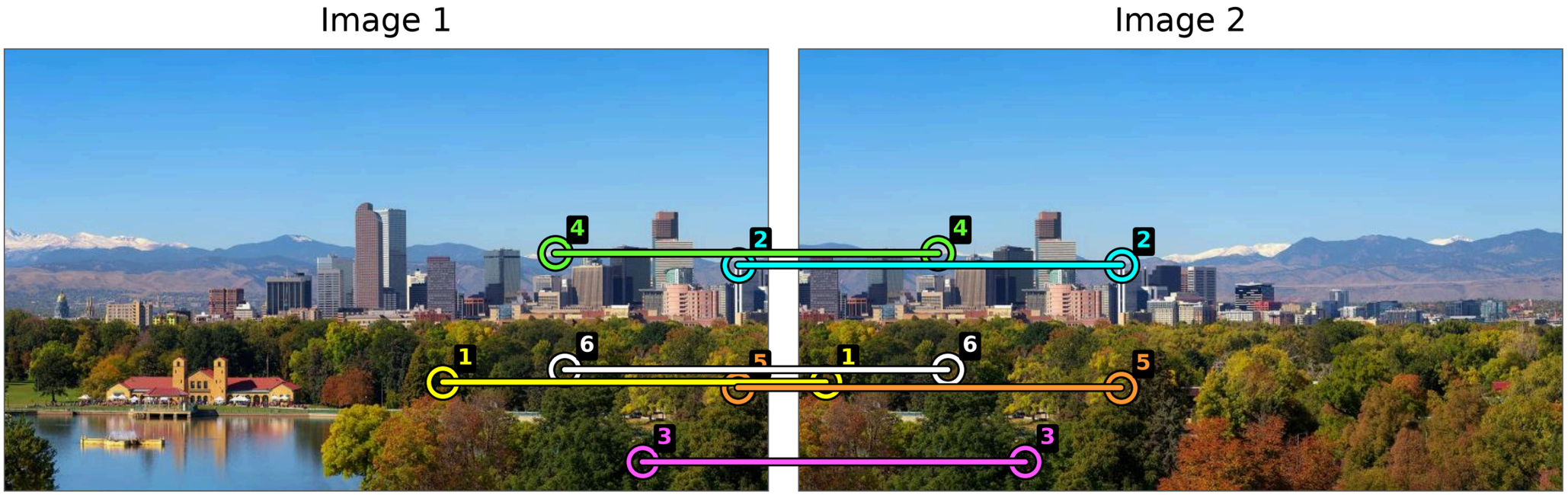
Image 2



Which points would you use to align these images?

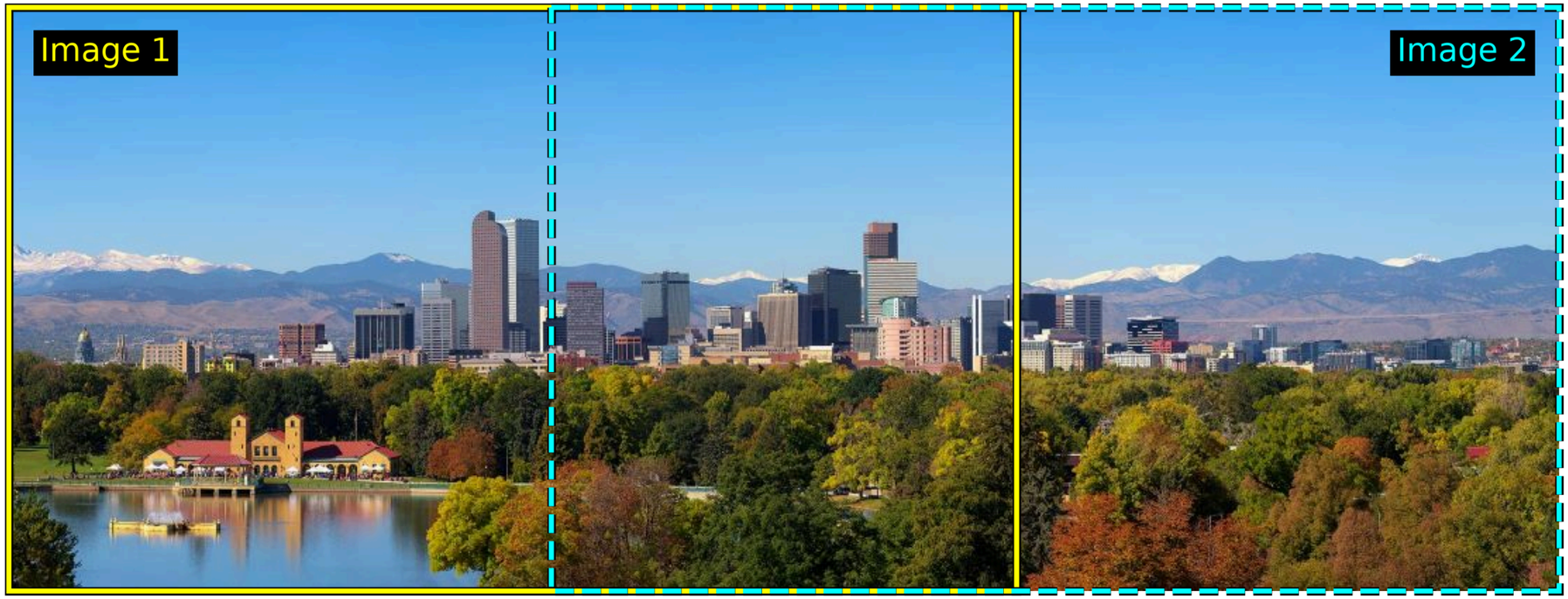
- Find recognizable details that appear in both views.
- Consider building corners, roof shapes, and other distinctive patterns.

Find Corresponding Points



- Each numbered pair identifies the same scene location in both images.
- These correspondences tell us how to align the two views.

Align the Images to Create a Panorama



- Aligning and combining the overlapping views recovers the original panorama.

What makes a point easy to recognize and find again?

What Is an Image Feature?

Image feature

A measurable visual property or pattern that helps us analyze or compare images.

- Features can include **edges**, **corners**, **blobs**, and **texture** patterns.
- In this lecture, we focus on **corners** as **local features**: small image regions that we can detect and match across images.
- A **keypoint** specifies the location of an interesting point.
- A **descriptor** summarizes the surrounding appearance as numbers for comparison.





The marked point provides a location. Its neighborhood provides an appearance we can compare.

Connection of Filtering, Edges, and Corners

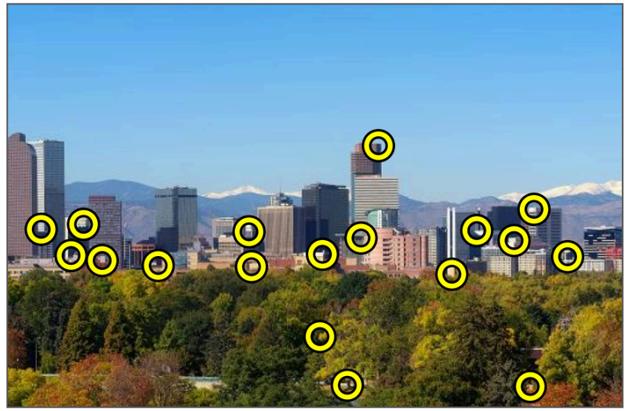
Filtering: smooth the image



Edges: gradient magnitude



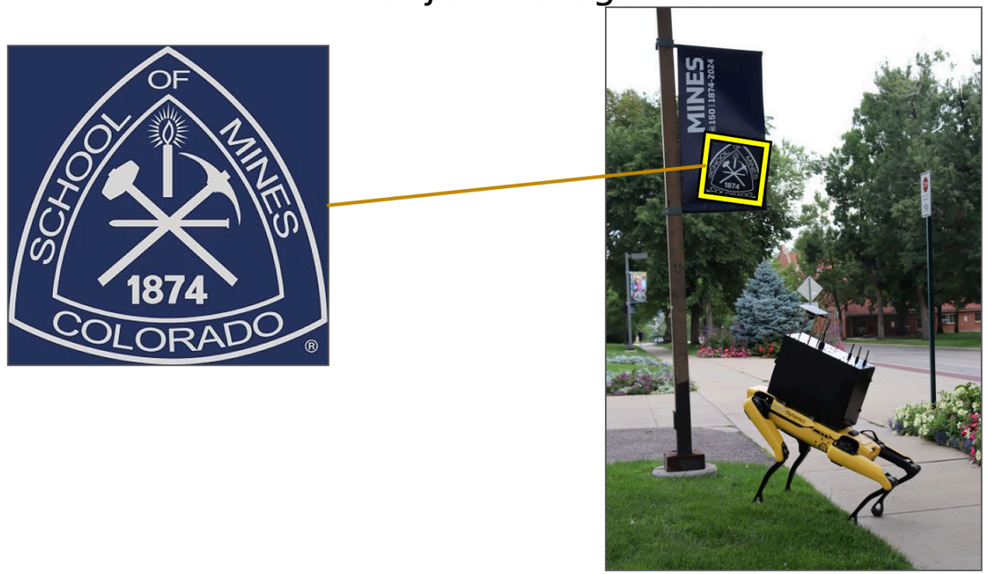
Corners: local edge structure



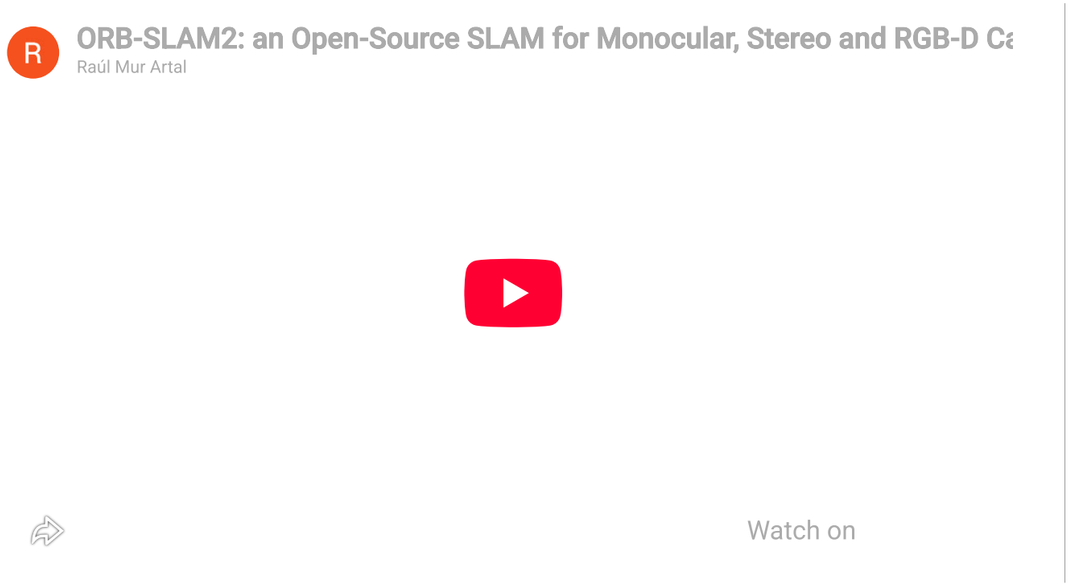
- **Filtering** gives us a tool to process images: smoothing to reduce noise, taking derivative to measure local intensity changes.
- **Edges**: Large gradient magnitudes indicate strong intensity changes in the image, primarily across an edge.
- **Corners**: Where edges meet or turn sharply, intensity changes in multiple directions within the neighborhood.

Applications of Image Features

Object recognition



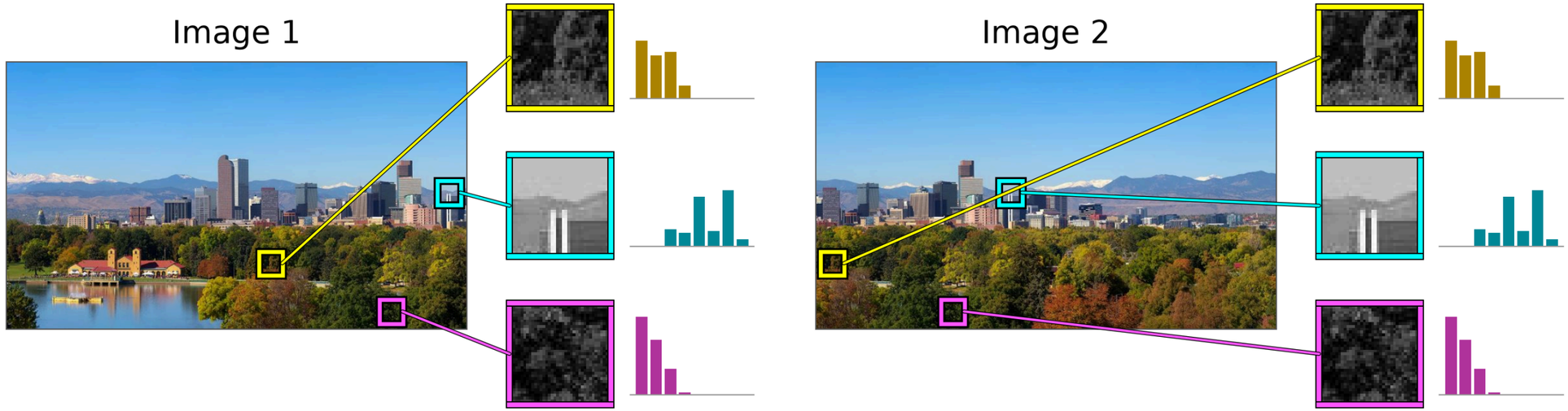
Localization and mapping



- **Image alignment**: match overlapping views to create panoramas.
- **Object recognition**: match a query image or object against stored images.
- **Motion tracking**: follow feature points across video frames.
- **3D reconstruction and localization**: track or match features across views to estimate scene structure and camera motion.

The common ingredient is finding the same scene points/features across images.

Main Components of Local Feature Matching



1. Detection

Find a set of distinctive features (aka, keypoints).

2. Description

Extract a feature descriptor around each interest point as a vector.

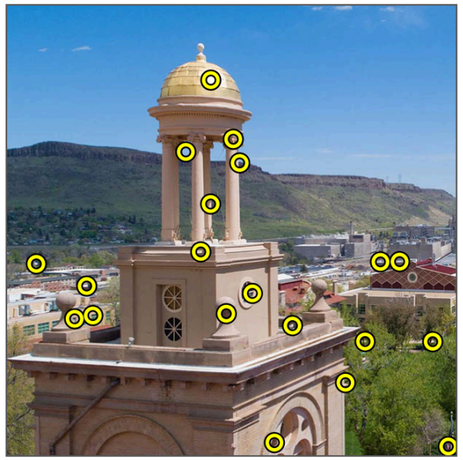
3. Matching

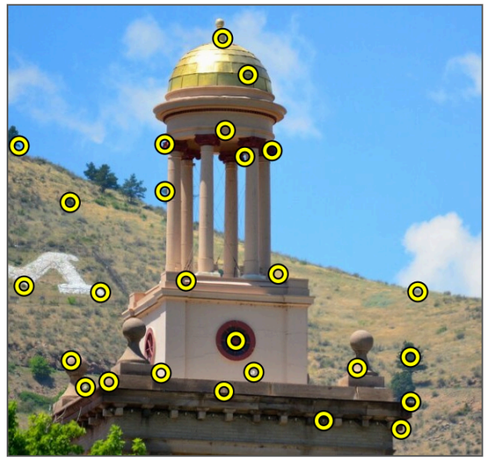
Compute distances between feature vectors to find correspondences.

What Makes a Good Feature?

A useful feature should be easy to find again, easy to distinguish from other features, and efficient to represent and compare.

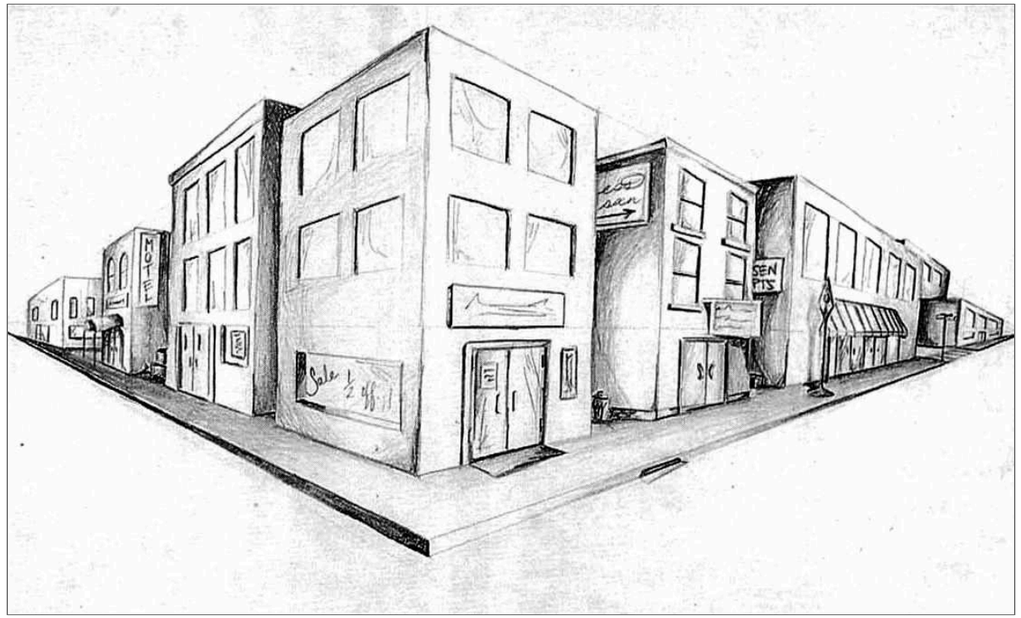
Property	What we want	Why it matters
Repeatability	Detect the same scene point across changes in viewpoint, scale, and lighting.	A point cannot be matched if we fail to detect it again.
Distinctiveness	A local appearance that differs from other candidate points.	Similar-looking regions can produce ambiguous matches.
Efficiency and compactness	A manageable number of keypoints with compact descriptors.	Detection, storage, and matching should remain practical.



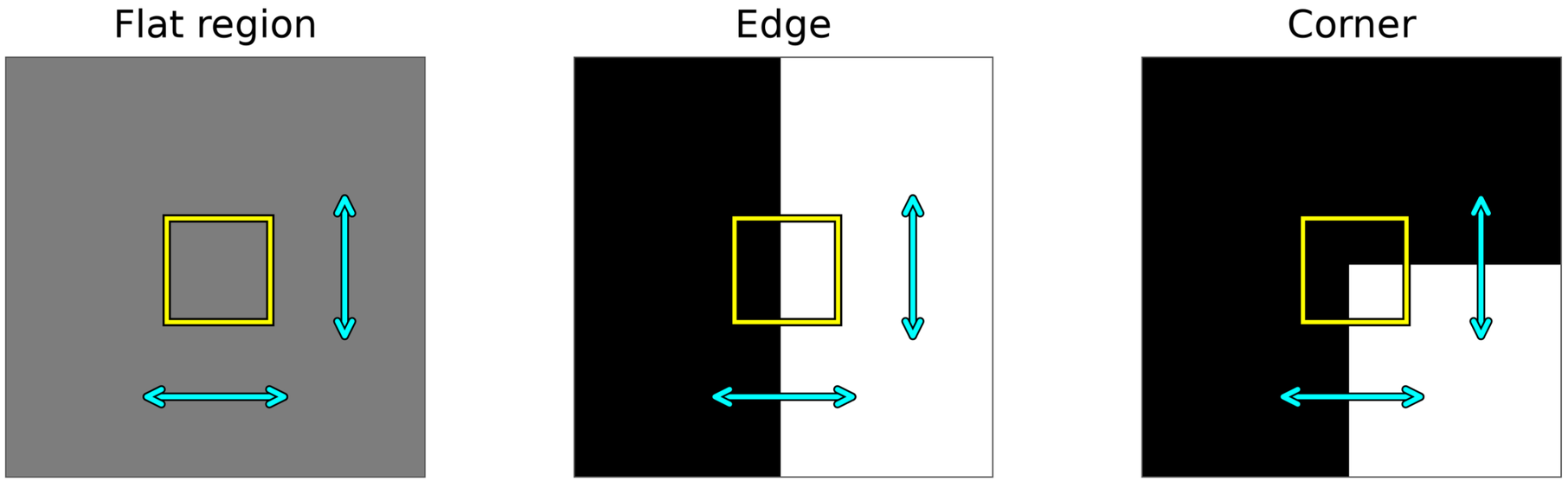


Corner Detection

1. Detection 2. Description 3. Matching



How Can We Detect Image Corners?



Imagine shifting each yellow window slightly and comparing its pixel values with the original patch.

- Along a straight edge, several nearby positions look similar.
- Around a corner, shifts in any direction change the patch, helping us locate it in both x and y.

Measuring Change with a Shifted Window

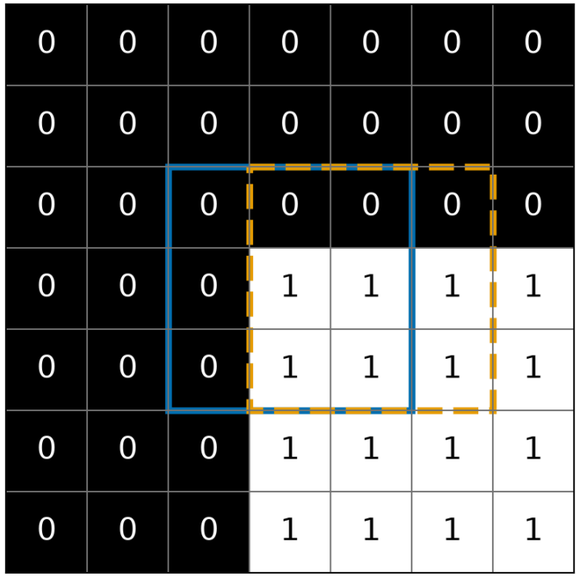
Keep a reference window W fixed and compare it with a shifted window in the same image.

$$E(u, v) = \sum_{(x,y) \in W} [I(x + u, y + v) - I(x, y)]^2$$

- u is the horizontal shift, and v is the vertical shift, in pixels.
- Positive u moves right; positive v moves down.
- Compare corresponding pixels, square their differences, and add them.

This defines the **auto-correlation** error $E(u, v)$: one number for each shift.

7 × 7 image



Blue: the reference 3×3 window. Orange: the window shifted one pixel right, $(u, v) = (1, 0)$.

One Shift, One Auto-correlation Value

Reference patch A

0	0	0
0	1	1
0	1	1

Shifted patch B

0	0	0
1	1	1
1	1	1

Subtract: B − A

0	0	0
1	0	0
1	0	0

Square: (B − A)²

0	0	0
1	0	0
1	0	0

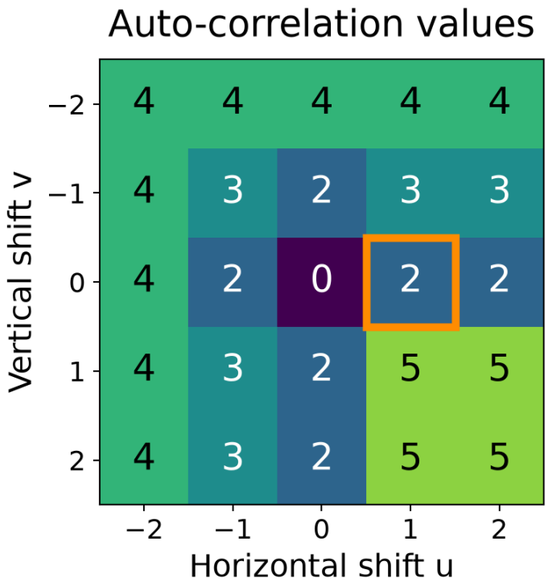
$$E(1,0) = \sum (B - A)^2 = 1 + 1 = 2$$

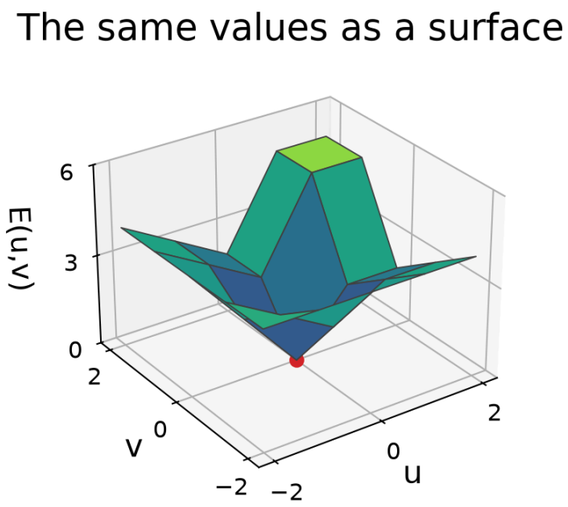
- Only two pixels change in this example; each contributes 1 to the sum.
- With no shift, every pixel matches itself: $E(0,0) = 0$.
- Squaring prevents positive and negative differences from canceling.

From Individual Shifts to the Response Map

Fixed reference window

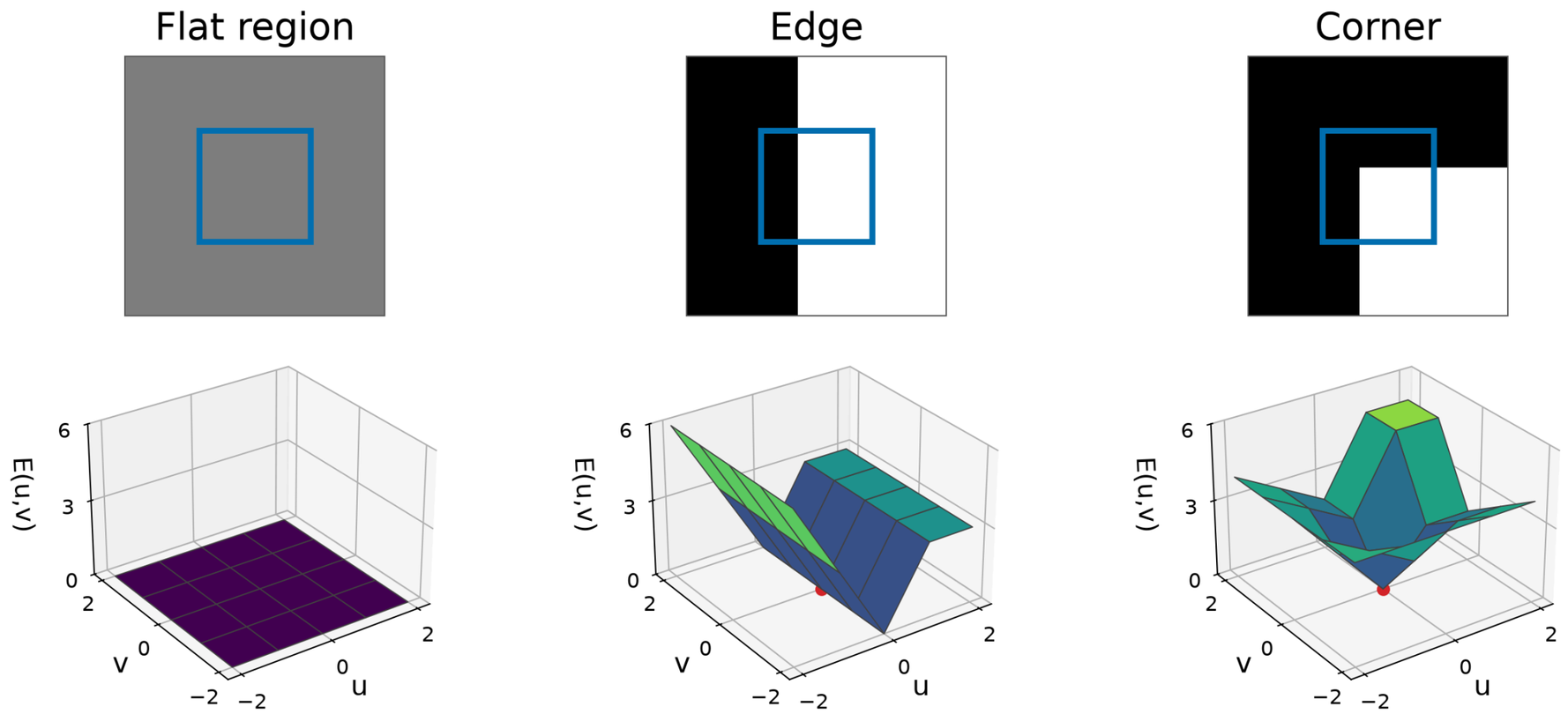
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	1	1	1
0	0	0	1	1	1	1
0	0	0	1	1	1	1
0	0	0	1	1	1	1





- Repeat the calculation for each shift: $u, v \in \{-2, -1, 0, 1, 2\}$.
- The orange cell is the previous result, $E(1, 0) = 2$. The center is $E(0, 0) = 0$.
- This map is indexed by **shifts**, not by image locations. The reference window stays fixed.

Flat, Edge, and Corner Response Surfaces



All three use the same 3×3 window, shifts, and vertical scale. The red point marks $(u, v) = (0, 0)$.

- **Flat region**: E stays zero. Edge: E stays low along the edge, forming a valley.
- **Corner**: E increases for small shifts in every direction, giving an isolated minimum.

Reading the Auto-correlation Response

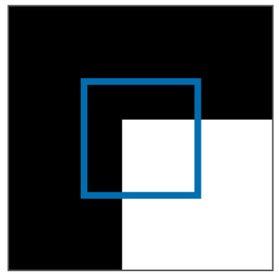
We compare a window with shifted versions of itself using the auto-correlation error

$$E(u, v) = \sum_{(x,y) \in W} [I(x + u, y + v) - I(x, y)]^2.$$

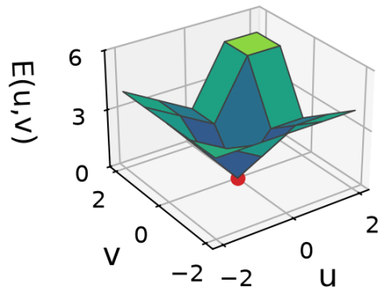
- At zero shift, the window matches itself, so $E(0, 0) = 0$.
- **Small values** mean that the shifted window looks **similar** to the reference window.
- **Large values** mean that the shifted window looks **different**.
- At a **corner**, $E(u, v)$ **rises strongly as we move away from zero in every direction**.

So, to find corners, we are interested in the shape of the response around $(u, v) = (0, 0)$.

Corner



Auto-correlation response



The Window Function

Uniform weights

1	1	1
1	1	1
1	1	1

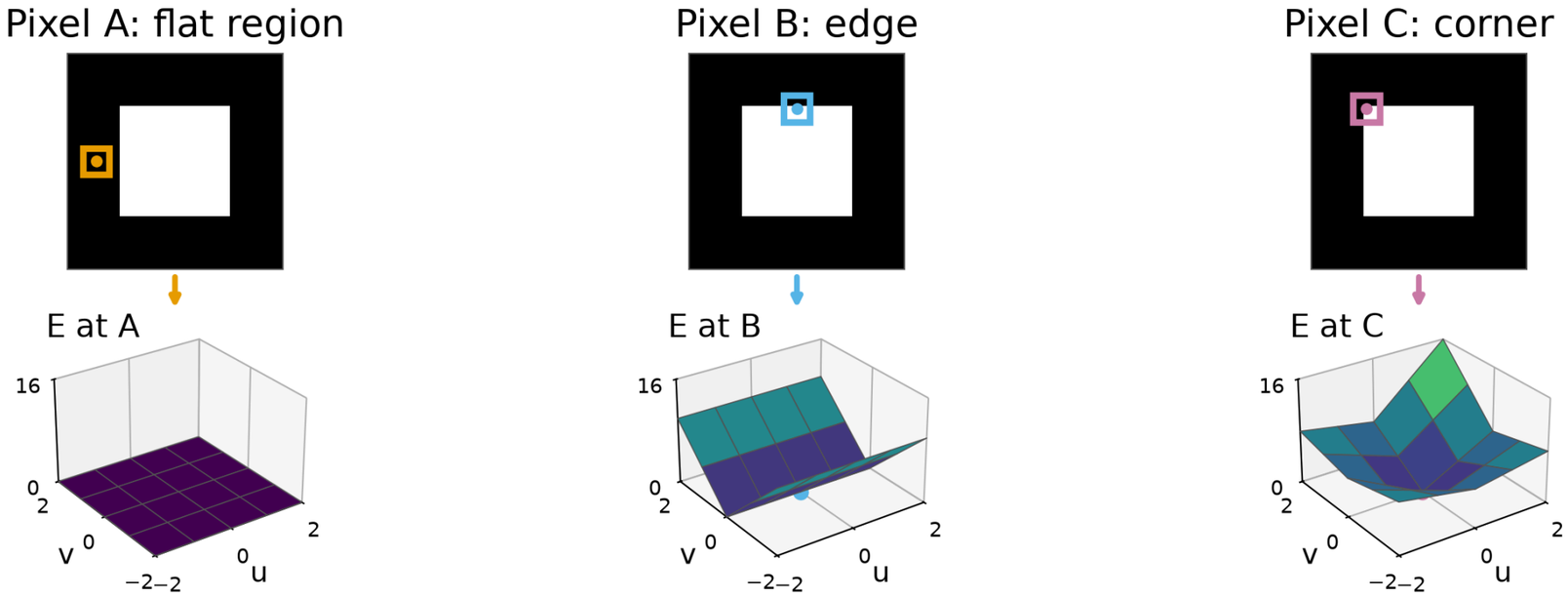
Gaussian weights: σ = 0.8

0.21	0.46	0.21
0.46	1.00	0.46
0.21	0.46	0.21

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u, y+v) - I(x,y)]^2$$

- Uniform weights count all pixels in the window equally. This is what we used so far.
- Gaussian weights emphasize the center and reduce the influence of pixels near the boundary.
- Weights are zero outside the window. The weights multiply the squared differences.

A Naive Corner-Detection Algorithm



1. Place a window around each pixel in the image.
2. Compute its auto-correlation $E(u, v)$ over a window.
3. Pick corners where $E(u, v)$ increases strongly in every shift direction.

Is this algorithm practical?

Naive Corner-Detection Computational Complexity

For a 1920 × 1080 pixels image, compute an auto-correlation response at every pixel:

Calculation	Count
Image pixels	2,073,600
Pixels in each window	$11 \times 11 = 121$
Nonzero shifts per window	$11 \times 11 - 1 = 120$
Pixel comparisons per image pixel	$121 \times 120 = 14,520$
Total pixel comparisons	**30.1 billion**

Assuming 10^8 pixel comparisons per second, this takes about **5 minutes for one image**.

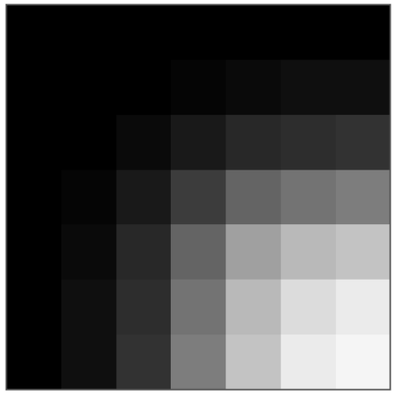
For an $H \times W$ image, a $k \times k$ window, and an $m \times m$ shift grid:

$$N_{\text{comparisons}} \approx \underbrace{HW}_{\text{image pixels}} \times \underbrace{k^2}_{\text{window pixels}} \times \underbrace{(m^2 - 1)}_{\text{nonzero shifts}}.$$

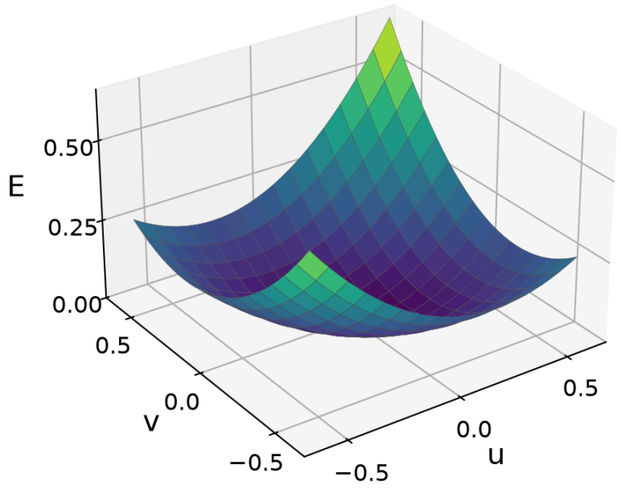
Computing auto-correlation at every pixel is impractical. **Can we approximate $E(u, v)$ without testing every shift?**

Approximating the Auto-correlation Response

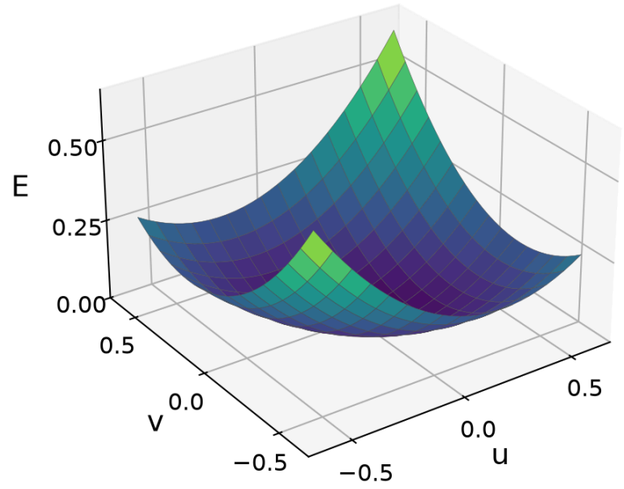
A smooth corner patch



Auto-correlation



Quadratic approximation



- Near zero shift, the auto-correlation response can be **approximated** by a **quadratic surface**.
- To detect corners, we need to know if the response stays low in any direction or rises in every direction.
- We will derive this surface from image gradients instead of computing auto-correlation.
- **Taylor expansion** connects auto-correlation shape to a small matrix of image gradients.

Recall: Taylor Series Expansion

- Taylor series around a reference point a for small h:

$$f(a+h) = f(a) + f'(a)h + \frac{f''(a)}{2!}h^2 + \frac{f'''(a)}{3!}h^3 + \cdots$$

- Keeping terms through h^2 gives a **quadratic approximation**.
- For $E(u, v)$ near $(0, 0)$, the corresponding approximation is

$$E(u,v) \approx E(0,0) + E_u u + E_v v + \frac{1}{2}\left(E_{uu}u^2 + 2E_{uv}uv + E_{vv}v^2\right).$$

- Derivative notations:

$$E_u = \frac{\partial E}{\partial u}, \qquad E_v = \frac{\partial E}{\partial v}.$$

- E_{uu} and E_{vv} are second derivatives; E_{uv} is the mixed derivative.
- All derivatives of E in this expansion are evaluated at $(u, v) = (0, 0)$.

Deriving the Quadratic Approximation

Define the intensity difference for a shifted window:

$$E(u,v) = \sum_{x,y} w(x,y)(\Delta I)^2, \quad \Delta I = I(x+u, y+v) - I(x,y).$$

1. Differentiate with respect to the shift:

$$\begin{aligned} E_u &= 2 \sum w \, \Delta I \, \Delta I_u, \\ E_v &= 2 \sum w \, \Delta I \, \Delta I_v, \end{aligned}$$

where $\Delta I_u = \partial(\Delta I)/\partial u$ and similarly for the other derivatives. The window weights stay fixed as we vary the shift.

Differentiate once more:

$$\begin{aligned} E_{uu} &= 2 \sum w \left[(\Delta I_u)^2 + \Delta I \, \Delta I_{uu} \right], \\ E_{uv} &= 2 \sum w \left[\Delta I_u \Delta I_v + \Delta I \, \Delta I_{uv} \right], \\ E_{vv} &= 2 \sum w \left[(\Delta I_v)^2 + \Delta I \, \Delta I_{vv} \right]. \end{aligned}$$

2. Evaluate at zero shift.

$$\Delta I = 0, \qquad \Delta I_u = I_x, \qquad \Delta I_v = I_y.$$

The constant and linear terms vanish:

$$E(0,0) = 0, \qquad E_u(0,0) = E_v(0,0) = 0.$$

The second derivatives simplify to

$$\begin{aligned} E_{uu}(0,0) &= 2 \sum w I_x^2, \\ E_{uv}(0,0) &= 2 \sum w I_x I_y, \\ E_{vv}(0,0) &= 2 \sum w I_y^2. \end{aligned}$$

Here, I_x and I_y are evaluated at the original pixel (x, y).

3. Substitute into the Taylor expansion.

$$\begin{aligned} E(u,v) &\approx \frac{1}{2} \left[2 \Big(\sum w I_x^2 \Big) u^2 + 4 \Big(\sum w I_x I_y \Big) uv + 2 \Big(\sum w I_y^2 \Big) v^2 \right] \\ &= \sum w \left(I_x^2 u^2 + 2 I_x I_y uv + I_y^2 v^2 \right). \end{aligned}$$

The Second Moment Matrix

The derivation gives a quadratic approximation:

$$E(u,v) \approx \sum_{x,y} w \left(I_x^2 u^2 + 2 I_x I_y uv + I_y^2 v^2 \right) = \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

The second moment matrix, also called the **structure tensor**, is

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} = \begin{bmatrix} A & B \\ B & C \end{bmatrix}.$$

Its three distinct entries are

$$A = \sum_{x,y} w I_x^2, \qquad B = \sum_{x,y} w I_x I_y, \qquad C = \sum_{x,y} w I_y^2.$$

- To compute M, first compute image derivatives I_x and I_y, then form these three weighted sums in each window.
- The matrix M describes how the window changes for small shifts in any direction.
- Next, we will connect the entries of M to the response shape and identify which shapes indicate corners.

Example: Compute M for a Small Image

Image I: levels 0 and 1

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	1	1	1
0	0	0	1	1	1	1
0	0	0	1	1	1	1
0	0	0	1	1	1	1

Full x derivative: I_x

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	0	1	1	0	0	0
0	0	1	1	0	0	0
0	0	1	1	0	0	0

Full y derivative: I_Y

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	1	1	1
0	0	0	1	1	1	1
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

- Find derivatives by filtering the image with $h_x = [-1, 0, 1]$ and $h_y = h_x^T$ kernels.
- The blue boxes show a 3×3 auto-correlation window.
- Next, form the gradient products and sum their values inside this window.

Example: Compute M for a Small Image

I_x^2

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	0	1	1	0	0	0
0	0	1	1	0	0	0
0	0	1	1	0	0	0

A = window sum = 4

I_xI_Y

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

B = window sum = 1

I_Y^2

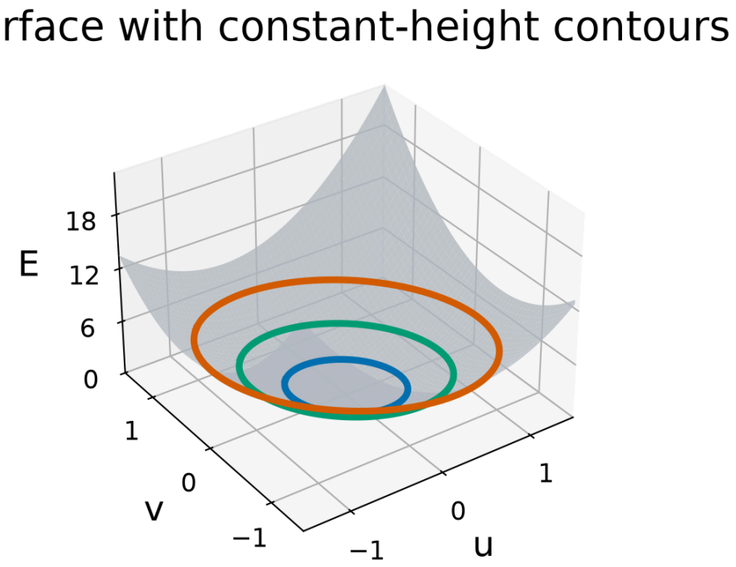
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	1	1	1
0	0	0	1	1	1	1
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

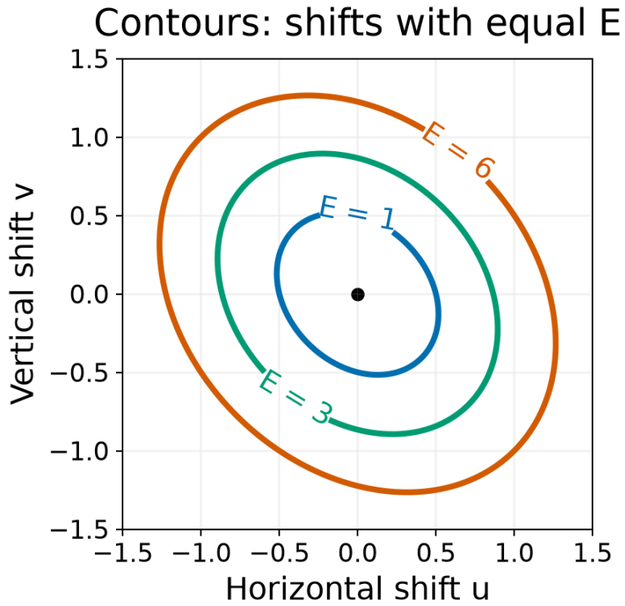
C = window sum = 4

$$A = \sum_{x,y} wI_x^2, \quad B = \sum_{x,y} wI_x I_y, \quad C = \sum_{x,y} wI_y^2 \quad \Longrightarrow \quad M = \begin{bmatrix} A & B \\ B & C \end{bmatrix} = \begin{bmatrix} 4 & 1 \\ 1 & 4 \end{bmatrix}.$$

- Assume $w = 1$ inside the 3×3 window. Sum only the values inside the window.
- A measures horizontal gradient energy, C vertical gradient energy, and B how the horizontal and vertical gradient components relate.

The Shape of the Quadratic Response

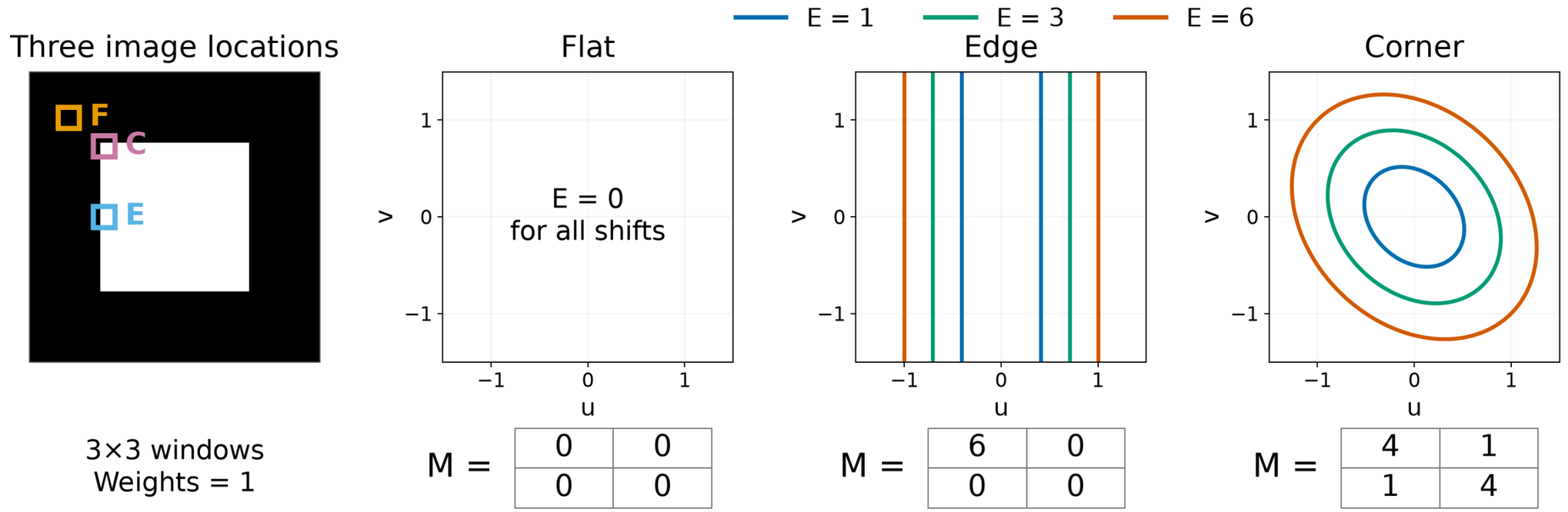




$$M = \begin{bmatrix} 4 & 1 \\ 1 & 4 \end{bmatrix} \quad \Rightarrow \quad E(u,v) \approx 4u^2 + 2uv + 4v^2.$$

- Each colored contour (or level set) joins u-v shifts with the same auto-correlation response.
- A nearby contour means a small shift already changes the patch substantially.
- This matrix gives closed contours: the response increases in every direction. Next, compare with a flat region or an edge.

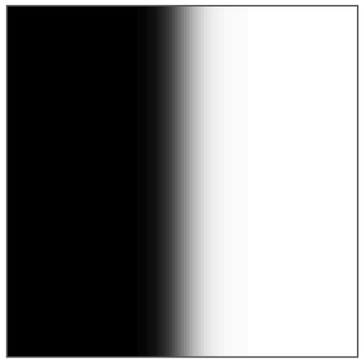
M Determines the Contour Shape

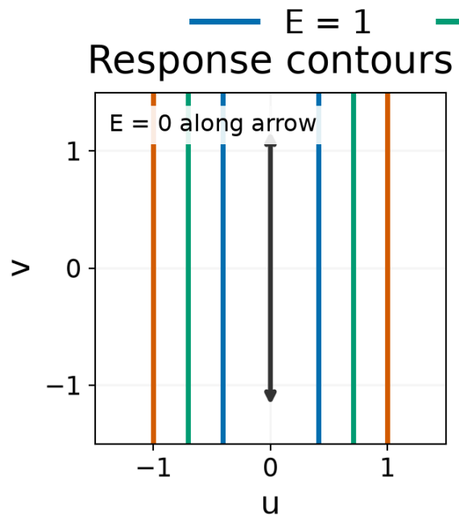


- **Flat:** $M = 0$, so the predicted response stays zero in every direction.
- **Edge:** the contours are parallel lines. Moving along them gives little change.
- **Corner:** the contours are closed ellipses. Moving away from zero in any direction changes the patch.
- **Can we guess the feature type directly from the elements of M?**

Rotation Changes the Entries of M

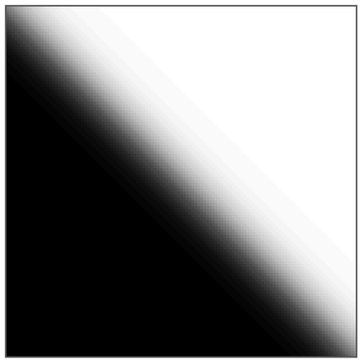
Vertical edge

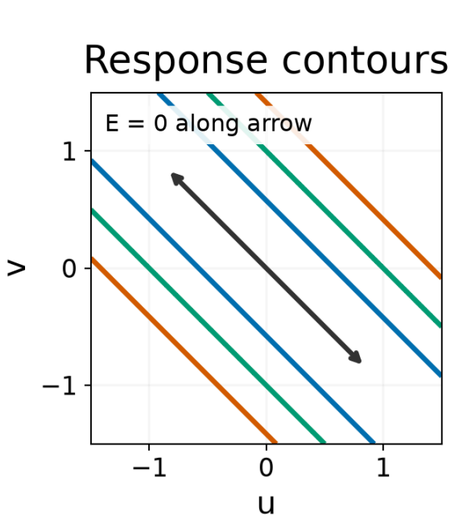




$$M = \begin{bmatrix} 6 & 0 \\ 0 & 0 \end{bmatrix}$$

Diagonal edge

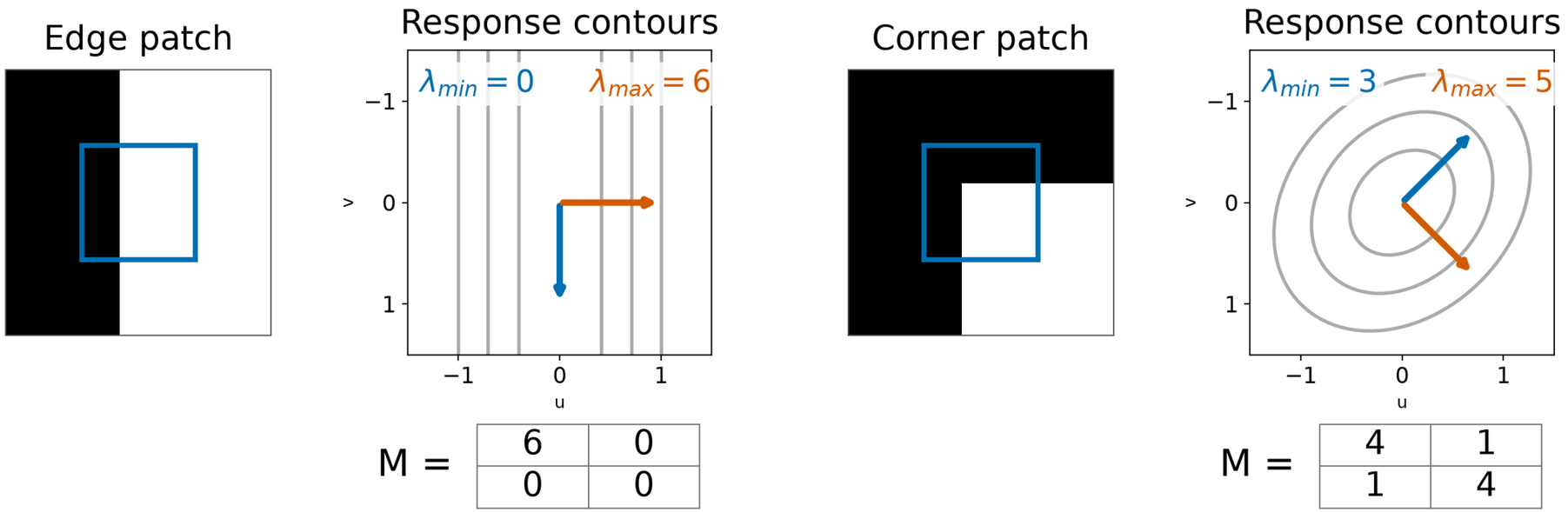




$$M = \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}$$

- Rotating an edge changes the entries of M without changing its feature type.
- For the diagonal edge, all diagonals are 3, but E stays zero along the arrow. Reading the entries alone can be misleading!
- We need the directions and magnitudes of least and greatest change.
- **Eigenvectors identify these directions. Eigenvalues measure the change along them.**

Eigenvalues Measure Change Along Principal Directions



M =

6	0
0	0

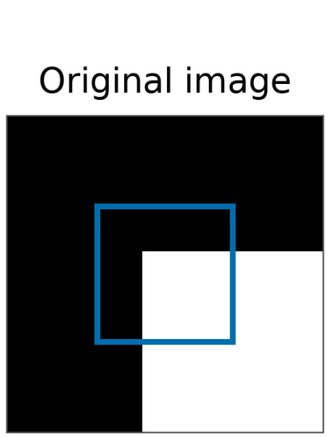
M =

4	1
1	4

$$M\mathbf{q}_i = \lambda_i \mathbf{q}_i \qquad (\|\mathbf{q}_i\| = 1).$$

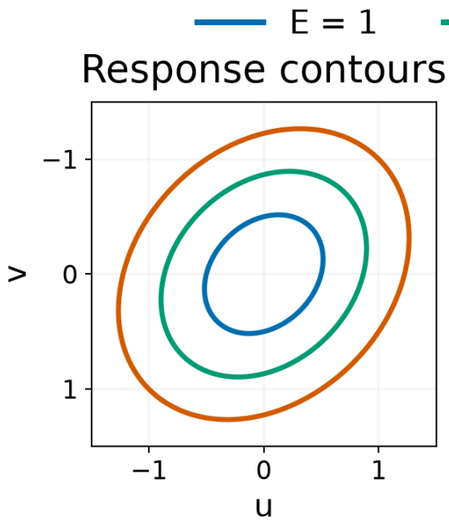
- λ_i is an eigenvalue, and $\mathbf{q}_i$ is an eigenvector.
- Orange: the perpendicular direction of greatest change, with strength $\lambda_{\max}$.
- Blue: the direction of least change, with strength $\lambda_{\min}$.
- The **edge has a zero eigenvalue**. The **corner has two large eigenvalues**.

Rotation Preserves the Eigenvalues



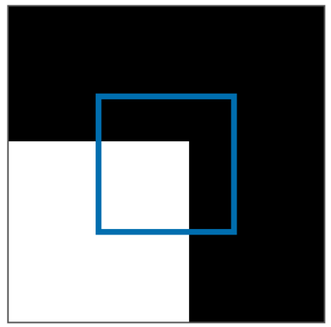
Original image

$\lambda_{min} = 3$
$\lambda_{max} = 5$



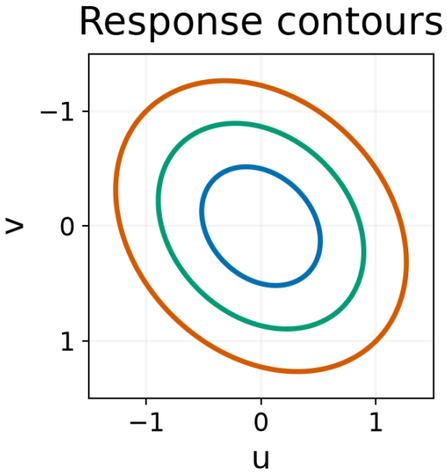
M =

4	1
1	4



Rotated 90° clockwise

$\lambda_{min} = 3$
$\lambda_{max} = 5$

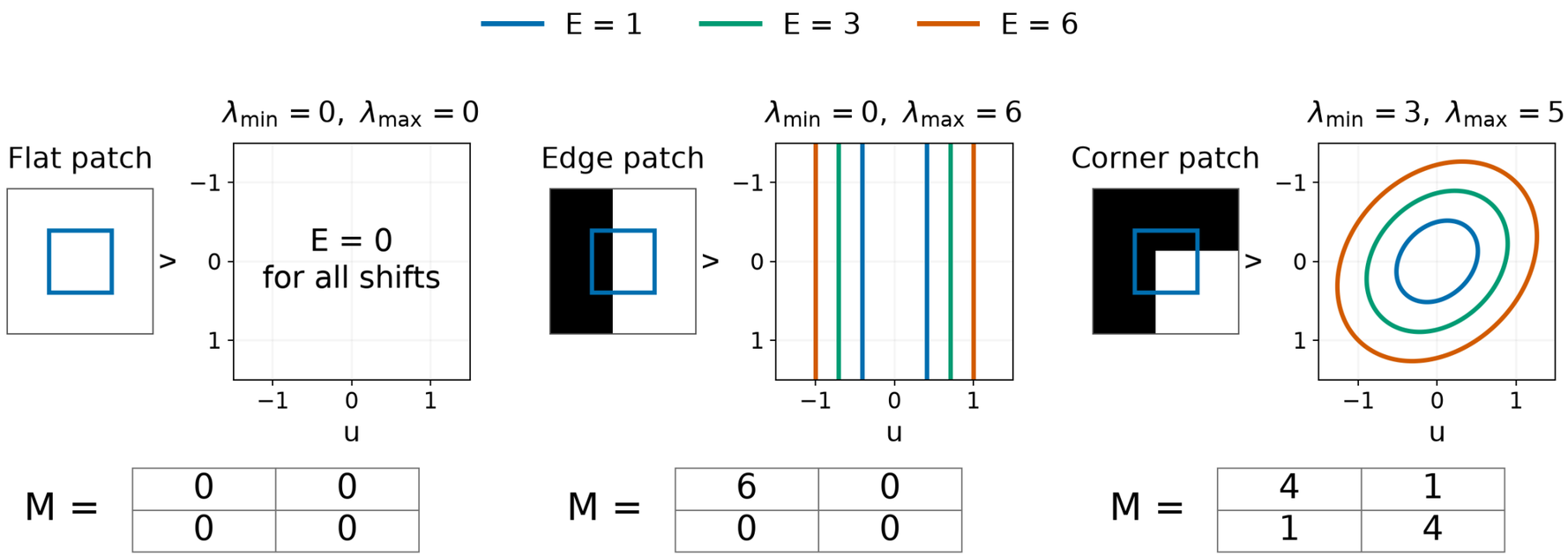


M =

4	-1
-1	4

- The plots show the response for an image and its rotated version.
- Their M matrices differ, but both have the same eigenvalues: $\lambda_{\min} = 3$ and $\lambda_{\max} = 5$.

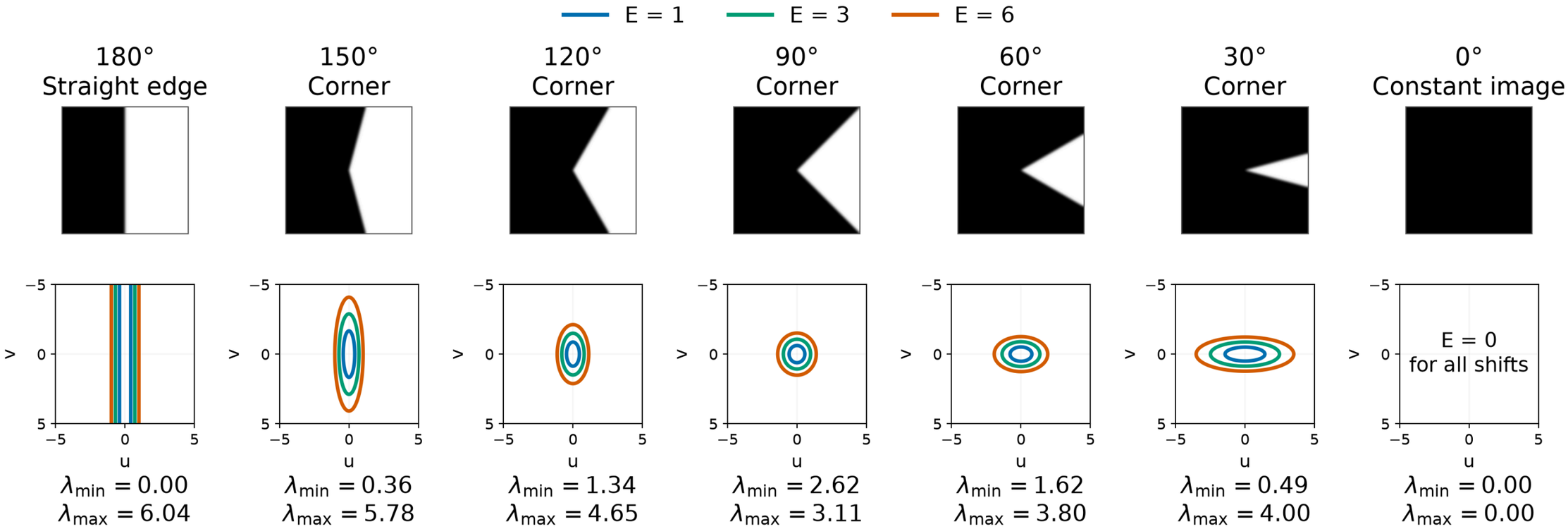
Reading Feature Type from Eigenvalues



Eigenvalues	Response shape	Feature type
Both small	Nearly flat in every direction	Flat region
One large, one small	Valley with a direction of little change	Edge
Both large	Bowl that rises strongly in every direction	Corner

Rotating a image patch changes its principal directions (eigenvectors), but preserves its eigenvalues and feature type.

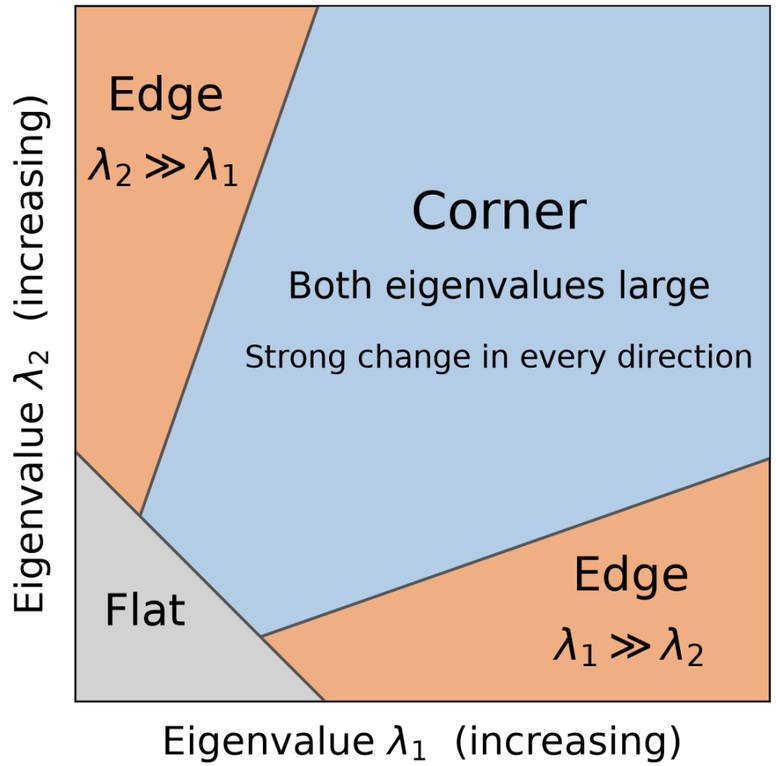
Corner Angle and Eigenvalues



- The 90° corner has the largest $\lambda_{\min}$: even its weakest direction changes strongly.
- Very acute or obtuse corners have elongated contours and a smaller $\lambda_{\min}$, making them more edge-like.

Classifying Image Points Using Eigenvalues of M

- Given λ_1 and λ_2 as eigenvalues of M.
- Each point in the diagram has coordinates (λ_1, λ_2).
- **Flat region:** both eigenvalues are small, so intensity changes little in any direction.
- **Edge:** one eigenvalue dominates, so change is strong across the edge and weak along it.
- **Corner:** both eigenvalues are large, so even the weakest direction has substantial change.

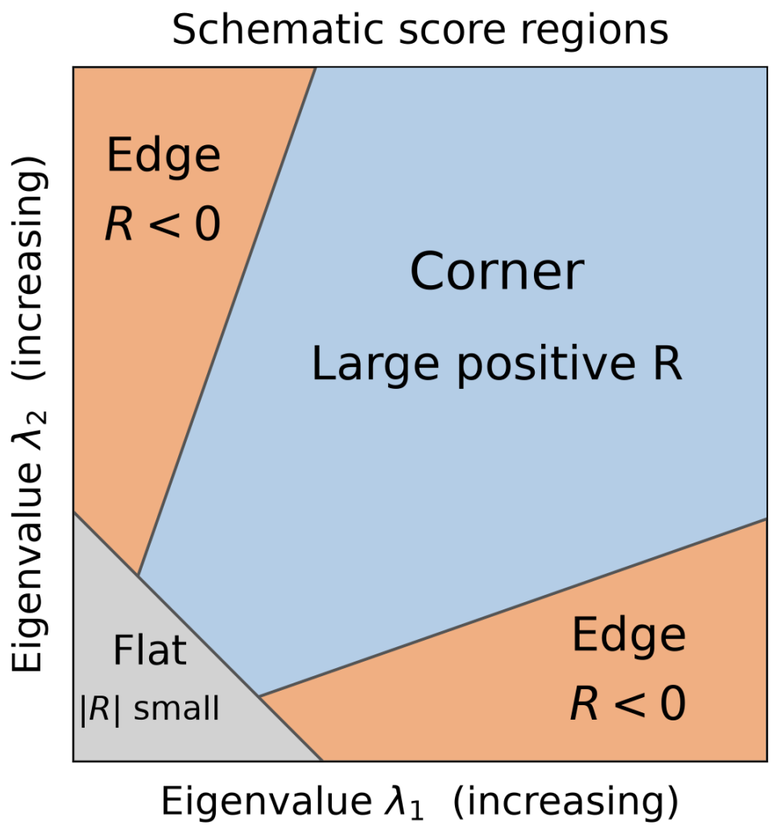


The Harris Corner Score

The **Harris corner score**, also called the **cornerness score**, combines both eigenvalues:

$$R = \lambda_1 \lambda_2 - \alpha(\lambda_1 + \lambda_2)^2.$$

- **Large positive R: corner candidate.**
- **Negative R: edge-like.**
- **Small $|R|$: weak response, often a flat region.**
- $\alpha = 0.04$ is a commonly used empirical value.
  - Equal eigenvalues: $R = (1 - 4\alpha)\lambda^2 = 0.84\lambda^2 > 0.$
  - Ideal edge ($\lambda_{\min} = 0$): $R = -0.04\lambda_{\max}^2 < 0.$
- Increasing α rejects more unequal eigenvalue pairs (and mark them as non-corners). It is a tuning parameter, not a universal optimum.



Cornerness Without Computing Eigenvalues

For any square matrix:

- The **determinant** is the **product of all its eigenvalues**.
- The **trace** is the **sum of its diagonal entries**, and also the **sum of its eigenvalues**.

For $M = \begin{bmatrix} A & B \\ B & C \end{bmatrix}$:

$$\det(M) = AC - B^2 = \lambda_1 \lambda_2, \qquad \operatorname{tr}(M) = A + C = \lambda_1 + \lambda_2.$$

The Harris cornerness score R can therefore be computed directly:

$$R = \lambda_1 \lambda_2 - \alpha(\lambda_1 + \lambda_2)^2 = \det(M) - \alpha \operatorname{tr}(M)^2 = (AC - B^2) - \alpha(A + C)^2.$$

Determinant and trace avoid explicit eigenvalue computation. For a 2×2 matrix, only a few multiplications and additions are needed, **reducing computation at every image pixel**.

Harris Corner Detector

Recall

$$M(x,y) = \begin{bmatrix} A & B \\ B & C \end{bmatrix}, \quad A = \sum_{x,y} w I_x^2, \quad B = \sum_{x,y} w I_x I_y, \quad C = \sum_{x,y} w I_y^2$$

$$R = \lambda_1 \lambda_2 - \alpha(\lambda_1 + \lambda_2)^2 = \det(M) - \alpha \operatorname{tr}(M)^2 = (AC - B^2) - \alpha(A + C)^2.$$

Harris corner detector Algorithm

Step	Operation	Output
1	Smooth the image, then compute image derivatives.	$I_x,\ I_y$
2	Form the gradient products at each pixel.	$I_x^2,\ I_x I_y,\ I_y^2$
3	Gaussian-average the products over a local window.	$A = g * I_x^2,\ B = g * (I_x I_y),\ C = g * I_y^2$
4	Compute the Harris score.	$R = AC - B^2 - \alpha(A + C)^2$
5	Thresholding: Keep pixels with a sufficiently large positive score.	$R > \tau$
6	Non-maximum suppression: Keep local maxima of the score.	Isolated corner locations

C. Harris and M. Stephens, “A Combined Corner and Edge Detector,” Alvey Vision Conference, 1988, pp. 147–151. (~23,000 citations.)

Harris Example

```
1 import cv2
2 import numpy as np
3
4 image_path = (
5     "assets/07_features/den_airport.jpg")
6 I_u8 = cv2.imread(
7     image_path, cv2.IMREAD_GRAYSCALE)
8 I = I_u8.astype(np.float32) / 255
```

- Which roof tips, window corners, and structural intersections should produce strong corner responses?
- Will a long, smooth roof edge give a strong corner response?

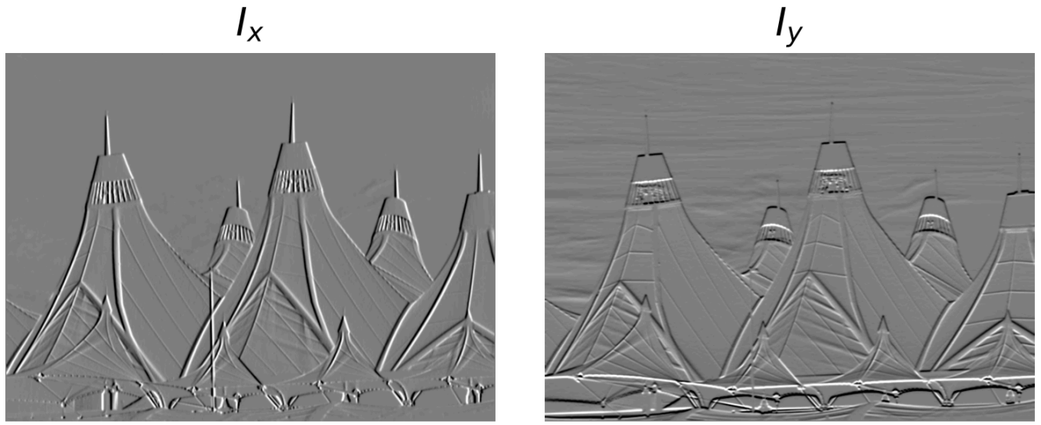


Step 1: Compute Image Gradients

```python
sigma_d = 1.0
J = cv2.GaussianBlur(I, (0, 0), sigma_d)

h_x = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1]
], dtype=np.float32) / 8

Ix = cv2.filter2D(J, -1, h_x)
Iy = cv2.filter2D(J, -1, h_x.T)
```



White and black indicate opposite derivative signs. Mid-gray indicates little change.

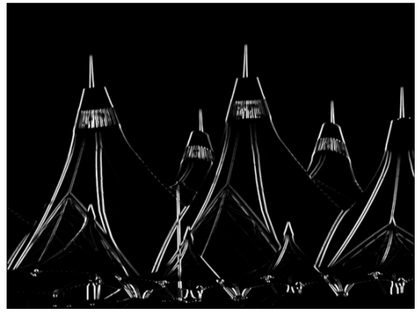
- Smoothing reduces the effect of noise before differentiation.
- I_x measures horizontal intensity change; I_y measures vertical intensity change.

Step 2: Form the Gradient Products

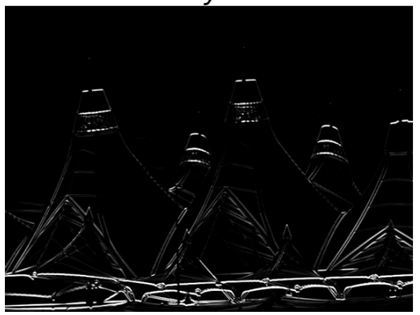
```
Pxx = Ix * Ix
Pxy = Ix * Iy
Pyy = Iy * Iy
```

- Squaring gives nonnegative horizontal and vertical gradient energies.
- $I_x I_y$ records how the two gradient components vary together. It can be positive or negative.

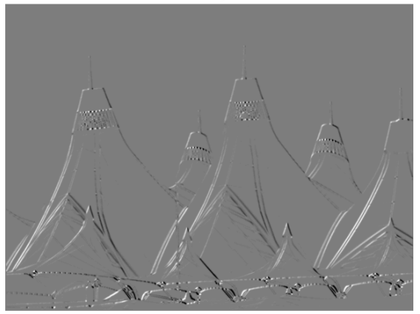
I_x^2



I_y^2



$I_x I_y$



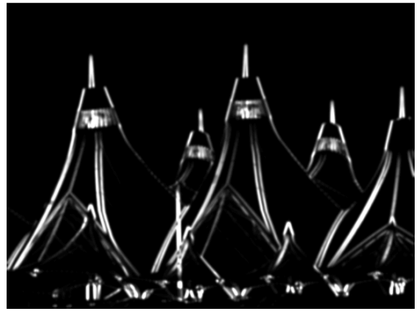
Step 3: Build the Second Moment Matrix

```
sigma_w = 2.0
A = cv2.GaussianBlur(Pxx, (0, 0), sigma_w)
B = cv2.GaussianBlur(Pxy, (0, 0), sigma_w)
C = cv2.GaussianBlur(Pyy, (0, 0), sigma_w)
```

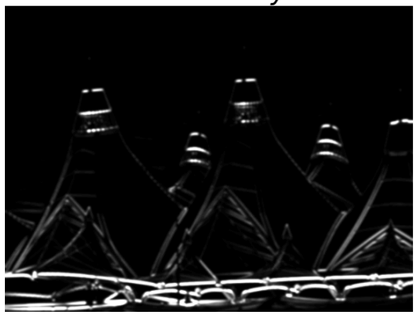
$$M(x, y) = \begin{bmatrix} A & B \\ B & C \end{bmatrix}.$$

- Gaussian filter combines gradient from neighboring pixels over the kernel window, e.g., to get $A = \sum_{x,y} w I_x^2$.
- σ_w sets the neighborhood size. It serves a *different purpose* from the initial smoothing scale σ_d.

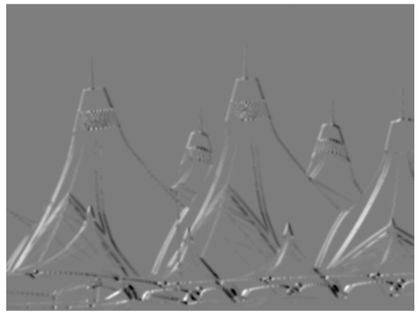
$A = g * I_x^2$



$C = g * I_y^2$



$B = g * (I_x I_y)$



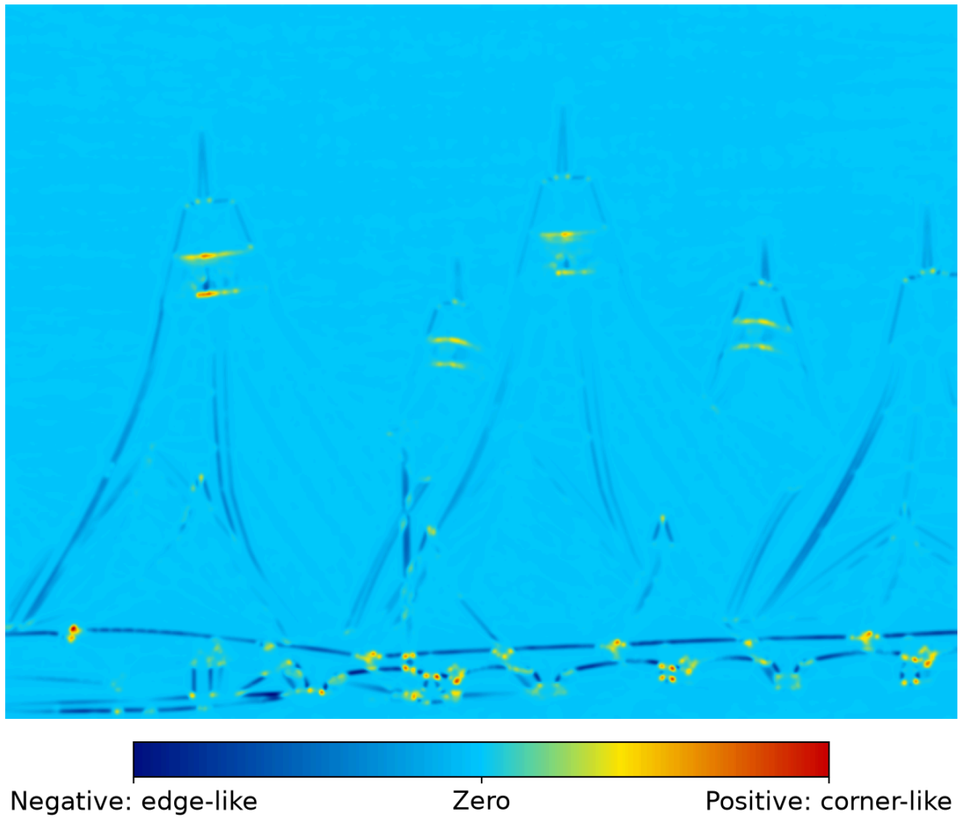
Step 4: Compute the Harris/Cornerness Score

```
alpha = 0.04

detM = A * C - B**2
traceM = A + C
R = detM - alpha * traceM**2
```

$$R(x, y) = \det(M) - \alpha \operatorname{tr}(M)^2.$$

- Large positive responses indicate corner candidates.
- Negative responses indicate a dominant edge direction.
- The sky has little gradient energy, so its response stays near zero.



Step 5: Threshold the Corner Response

```
quality = 0.01
threshold = quality * R.max()
candidates = (R > 0) & (R > threshold)
```

- Keep only sufficiently strong positive responses.
- Increasing the threshold removes weaker candidates.
- One corner often produces a cluster of selected pixels. Thresholding alone does not choose a single location.

16,188 candidate pixels



Step 6: Non-Maximum Suppression

```
window = np.ones((13, 13), dtype=np.uint8)
local_max = cv2.dilate(R, window)
corners = candidates & (R == local_max)
y, x = np.nonzero(corners)
```

- Dilation gives the largest score in each 13×13 neighborhood.
- Keep a candidate only if its score equals the neighborhood maximum.
- A larger neighborhood suppresses nearby peaks more strongly.
- The selected points concentrate near roof tips, window corners, and structural intersections.

178 local maxima after suppression



Harris Corner Detection in OpenCV

```
R_cv = cv2.cornerHarris(
    I, blockSize=5, ksize=3, k=0.04)

threshold = 0.01 * R_cv.max()
window = np.ones((13, 13), dtype=np.uint8)
local_max = cv2.dilate(R_cv, window)
corners_cv = ((R_cv > threshold)
              & (R_cv == local_max))
y_cv, x_cv = np.nonzero(corners_cv)
```

- `blockSize=5`: combine gradient products over a 5×5 neighborhood to form M.
- `ksize=3`: use 3×3 Sobel filters to compute image derivatives.
- `k=0.04`: the Harris parameter α in $R = \det(M) - \alpha \operatorname{tr}(M)^2$.

216 corners after thresholding and suppression

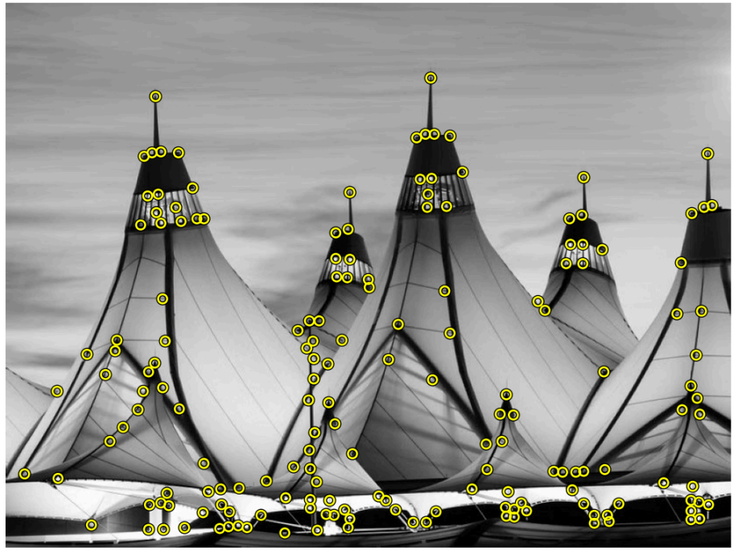


`cornerHarris` returns a score at each pixel. We then apply thresholding and non-maximum suppression.

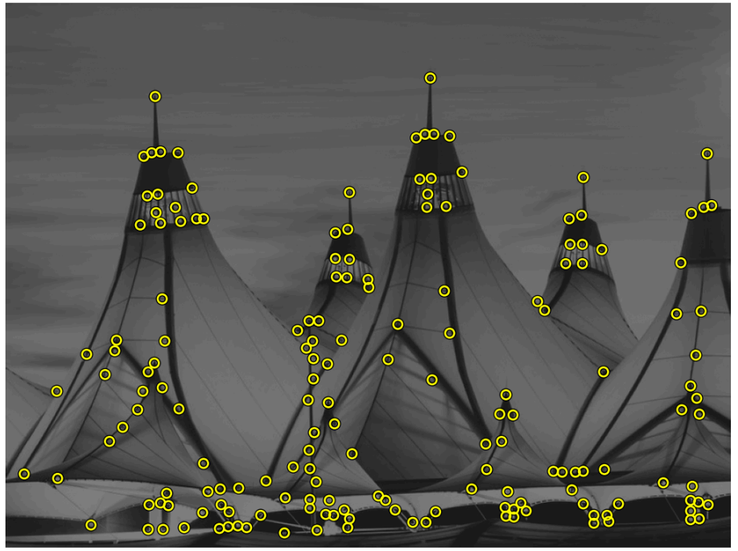
Invariance and Covariance

- **Invariance:** changing image intensities should leave the detected **locations unchanged**.
- **Covariance (equivariance):** moving the image should move the detected points **with it**.

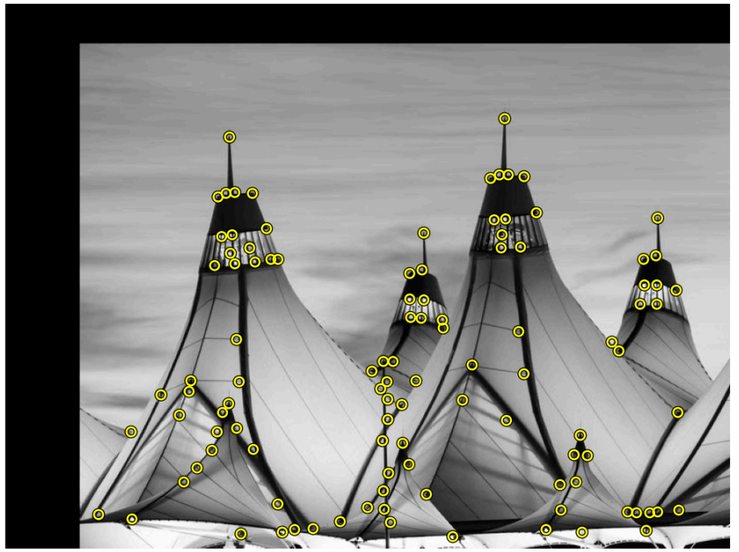
Original



Intensity change: Invariance



Translation: Covariance

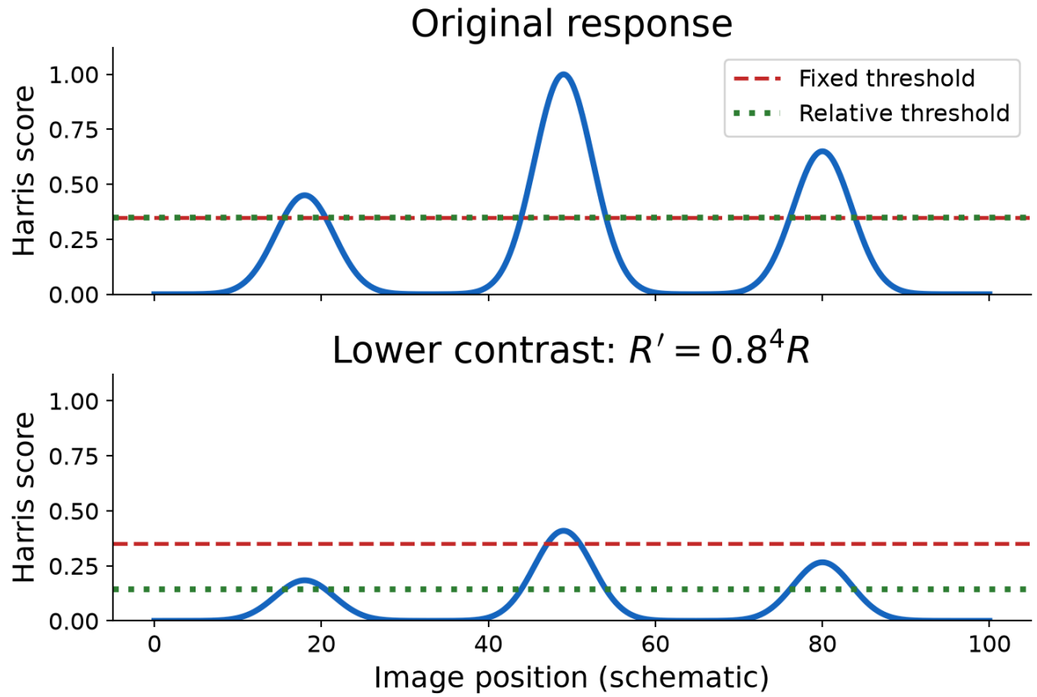


Harris Under Affine Intensity Changes

For $I' = aI + b$, with constant $a > 0$ and b:

$$\nabla I' = a\nabla I, \qquad M' = a^2 M,$$

$$\boxed{R' = a^4 R}.$$



- **Brightness offset** b**:** derivatives and scores stay unchanged.
- **Contrast gain** a**:** peak locations stay unchanged, but score magnitudes change.
- A **fixed threshold** can select different corners.

- A **relative threshold** $R > q\max(R)$ scales with the response and preserves the selection in this ideal model.
- But clipping, quantization, noise, and spatially varying illumination can break this ideal behavior.

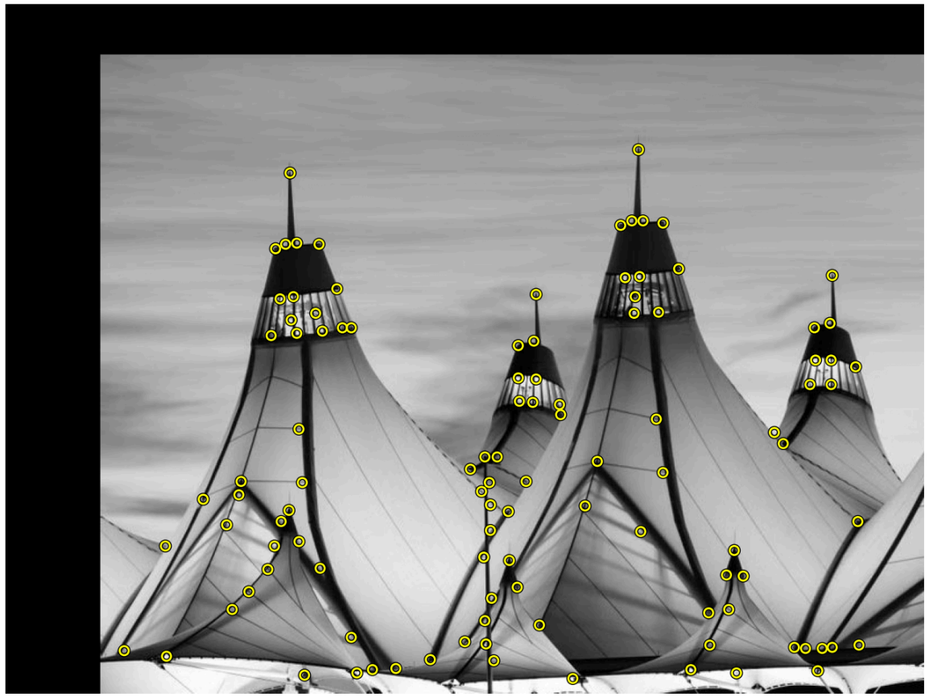
Harris Under Image Translation

For a translation $\mathbf{t} = (\Delta x, \Delta y)^T$, corresponding corners move from $\mathbf{x}$ to $\mathbf{x} + \mathbf{t}$.

Original



Translated: $\Delta x = 85$, $\Delta y = 45$ pixels



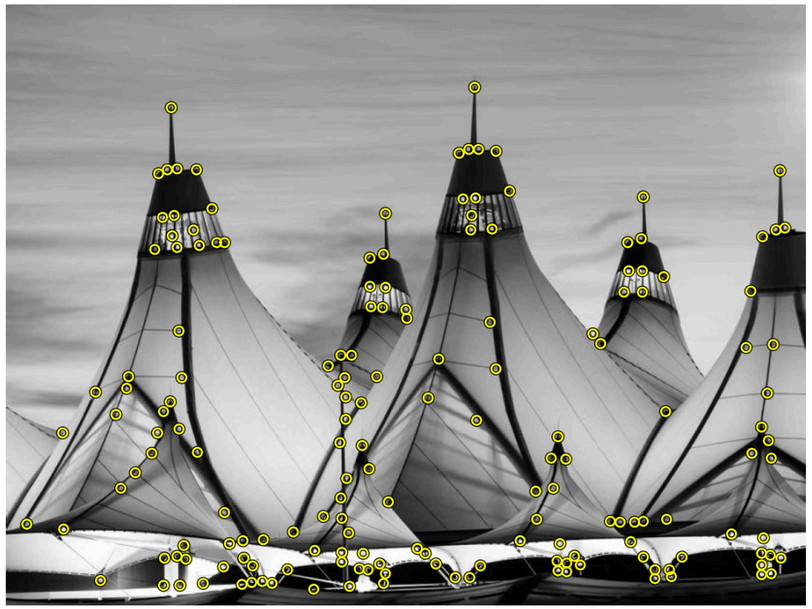
- The derivative filters and local averaging window apply the **same operation at every location**.
- The response map shifts with the image: $R'(\mathbf{x} + \mathbf{t}) = R(\mathbf{x})$.

Harris Under Image Rotation

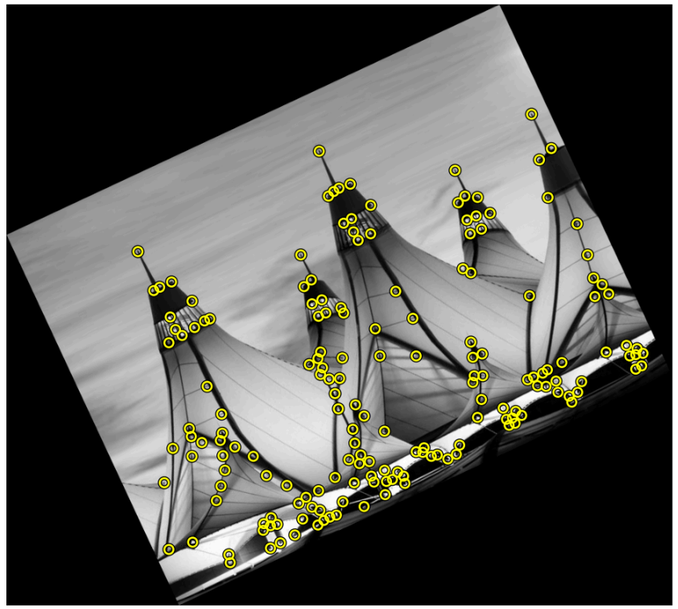
For a rotation matrix Q, the second moment matrix changes orientation while **its eigenvalues and Harris score stay unchanged**.

$$M' = QMQ^T \quad \Longrightarrow \quad \lambda'_1 = \lambda_1,\ \lambda'_2 = \lambda_2,\ R' = R.$$

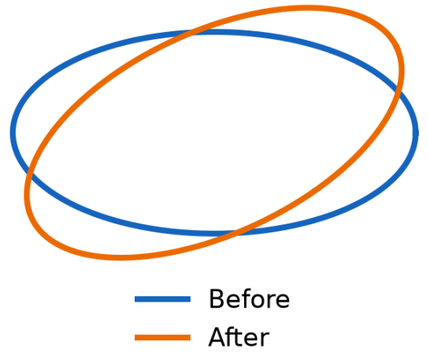
Original



Rotated 25 degrees



Same contour shape

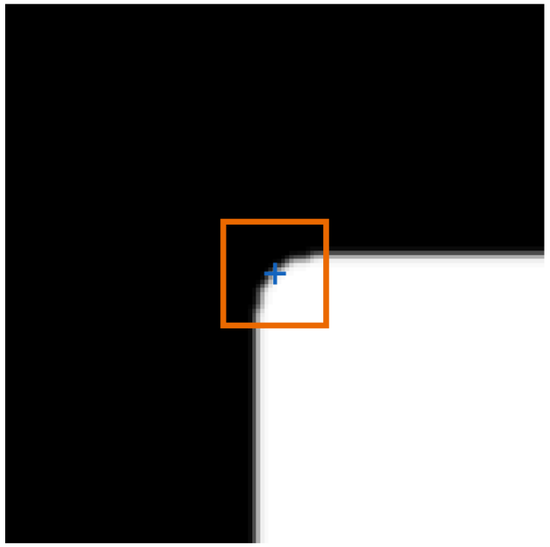


- **Corner locations are covariant:** the points rotate with the image.

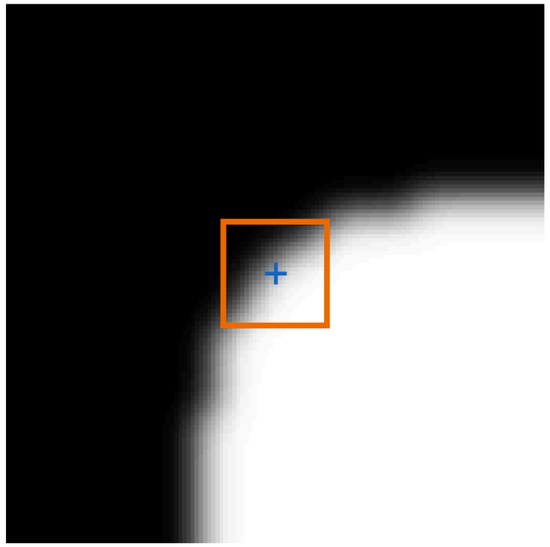
Harris Under Scaling

A window that contains **several gradient directions** at one scale may see **only a short, nearly straight edge** after enlargement.

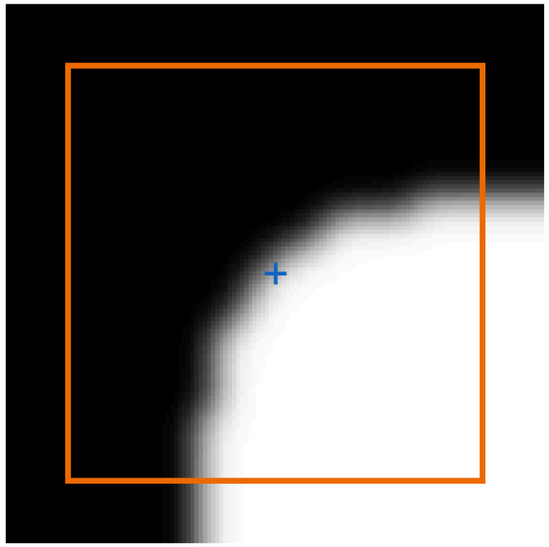
Small structure, 25-pixel window



4x larger, same 25-pixel window



4x larger, 100-pixel window

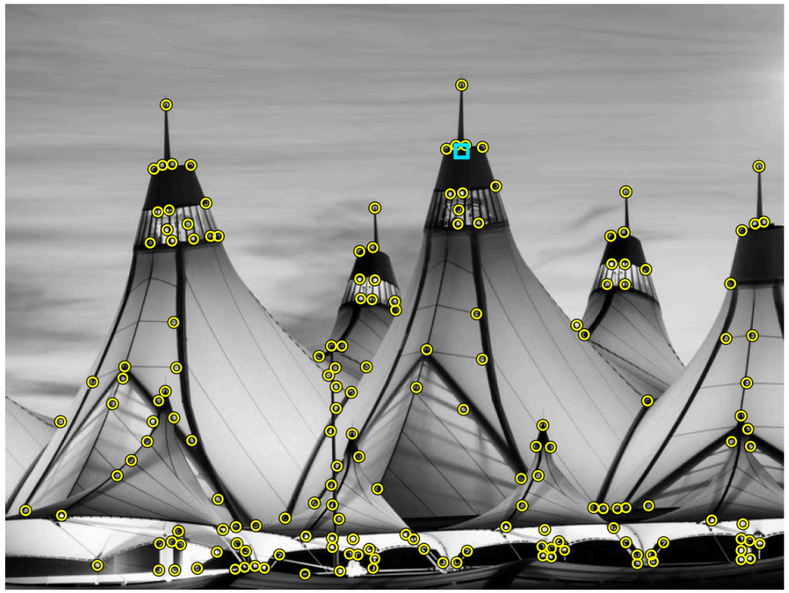


- **Harris with a fixed window size is not scale covariant.** Resizing the image can change which corners are detected.
- To handle scale changes, we need to adjust the smoothing scale and neighborhood size.

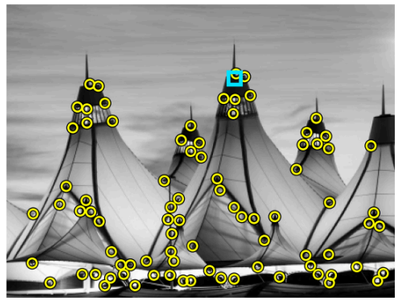
Scaling the Airport Image

We halve the image width and height, then rerun Harris with unchanged settings:

Original: 170 corners



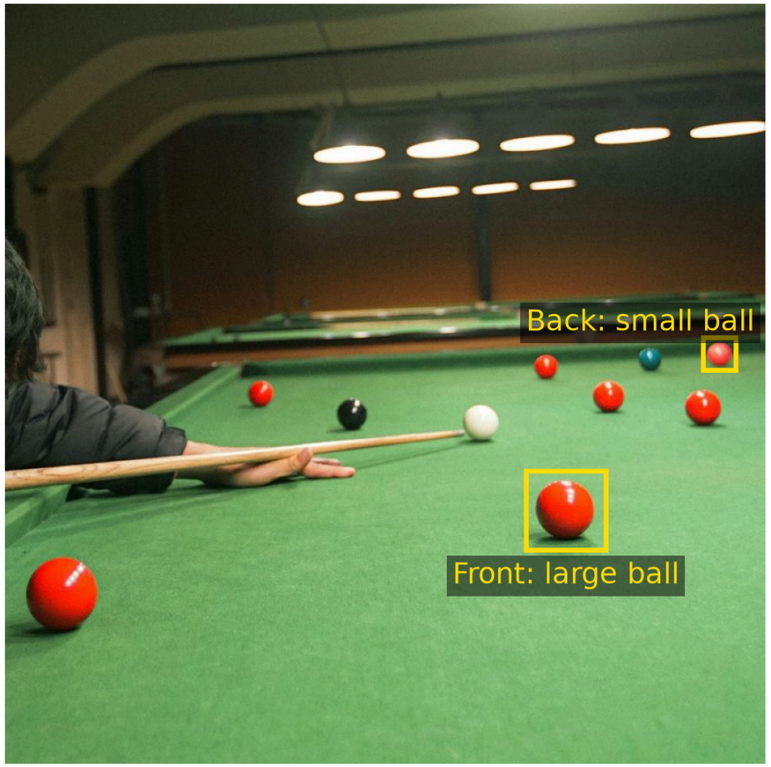
Half size: 84 corners



- Cyan boxes: the same 13×13 max-suppression window. It covers more of the scene in the smaller image.
- **Different corners survive:** shrinking changes image detail, and the fixed window suppresses more nearby peaks.
- **Harris is not scale covariant.** We need to detect features across multiple scales.

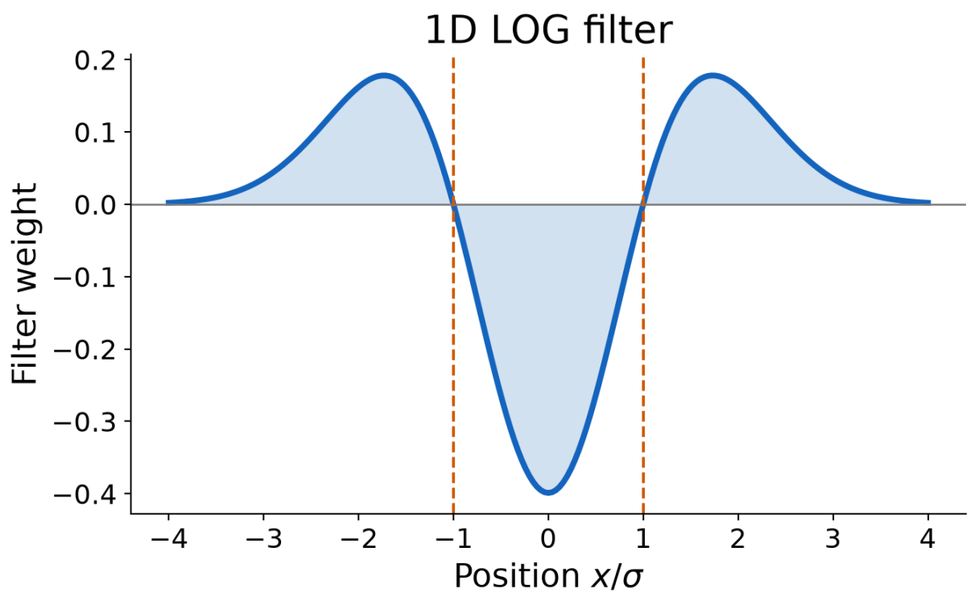
Multi-Scale Detection

Automatic Scale Selection

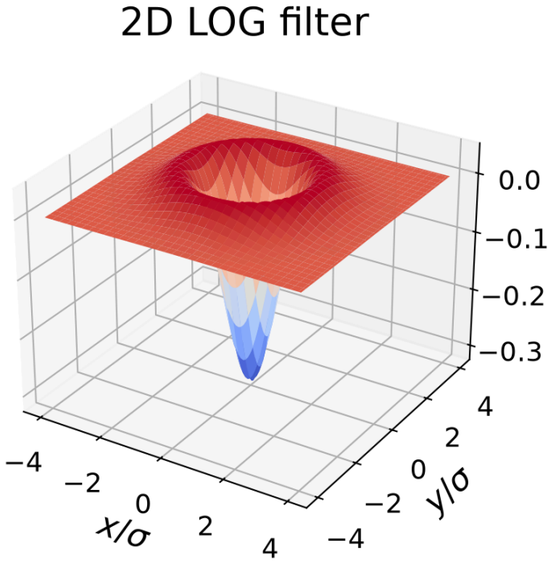


- How can we make a feature detector scale-invariant?
- How can we automatically select the scale?
- Which filter size gives each ball its strongest response?

Recall: Laplacian-of-Gaussian Filter



$$K_\sigma^{1D} = \sigma^2 \frac{d^2 G_\sigma}{dx^2}$$

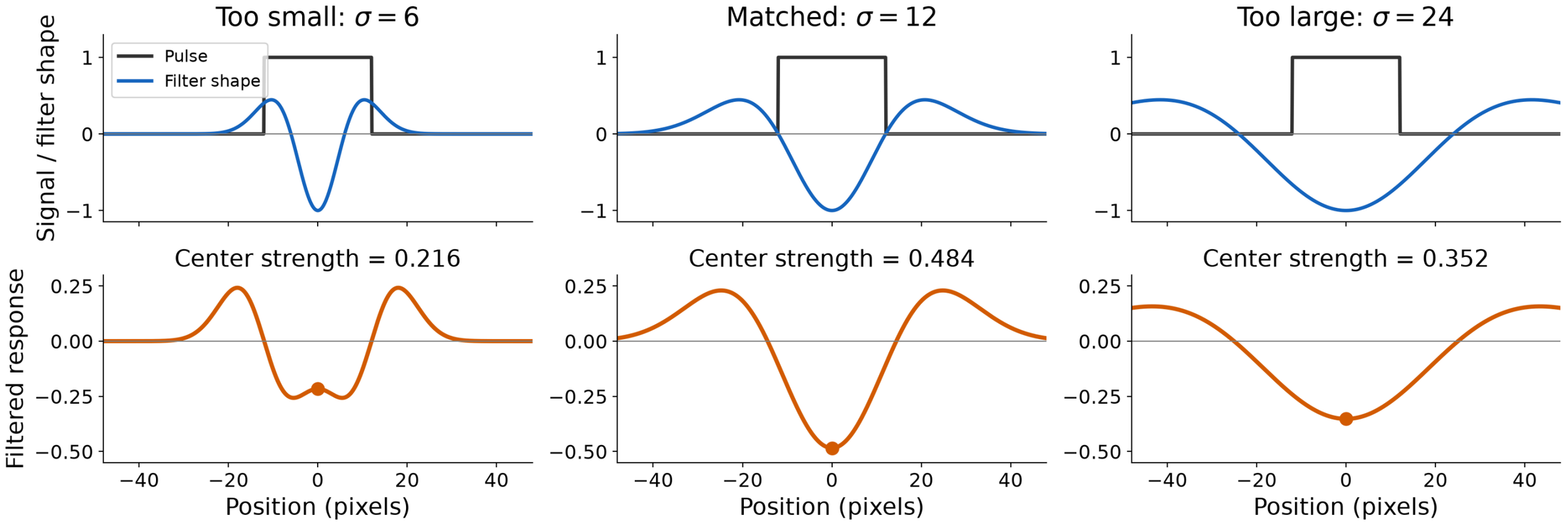


$$K_\sigma^{2D} = \sigma^2 \nabla^2 G_\sigma$$

- The LOG filter is the second derivative of a Gaussian kernel.
- The scale σ controls the filter size.

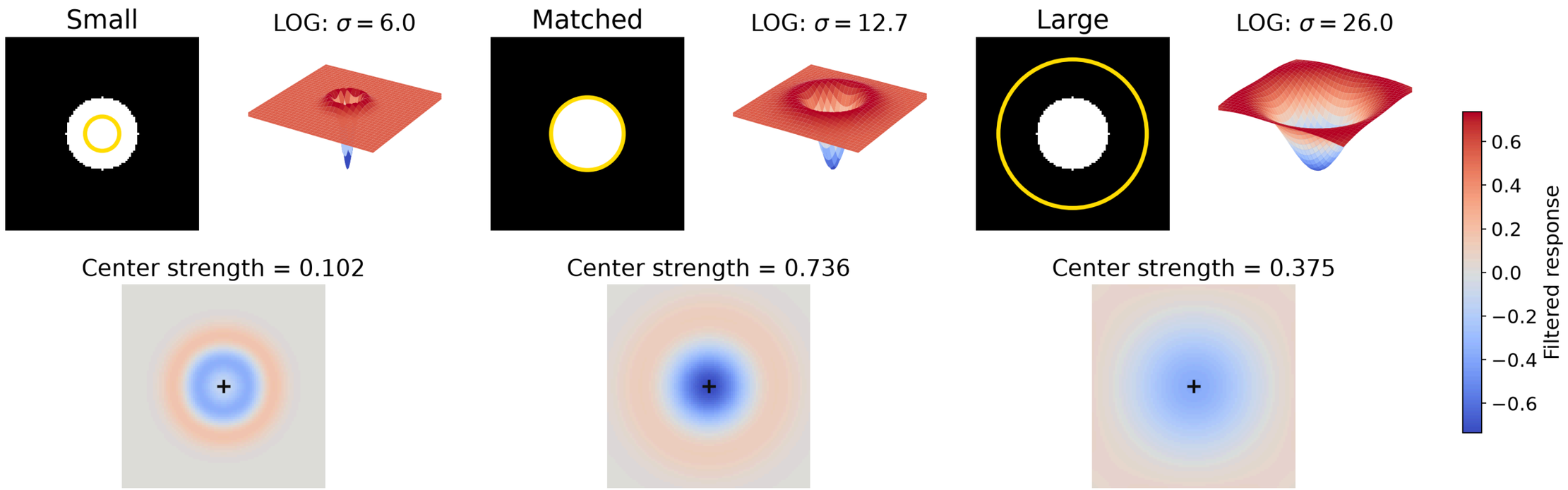
Filter Response vs. Signal Size

The response at the **pulse center** is strongest when the filter's central lobe fits the pulse.



- The pulse width is 24 pixels. At $\sigma = 12$, the filter's zero crossings at $\pm\sigma$ match its edges.
- The highest response (in magnitude) is achieved when the pulse width matches the filter zero crossings.

Filter response vs. Feature Size



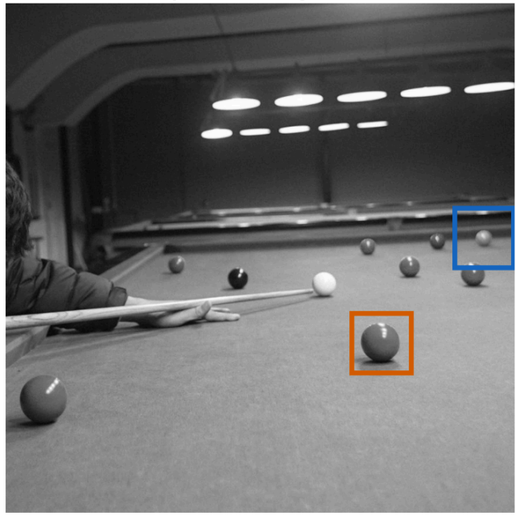
- The same observation holds for a circular feature in a 2D image filtered by a LOG kernel.
- $R_\sigma = K_\sigma^{2D} * I$, and response strength $= |R_\sigma|$
- The yellow circle marks the filter's zero crossing, $r = \sqrt{2}\sigma$.
- **The strongest response occurs when filter zero crossing fits the disk.**

Filter Response at Different Scales

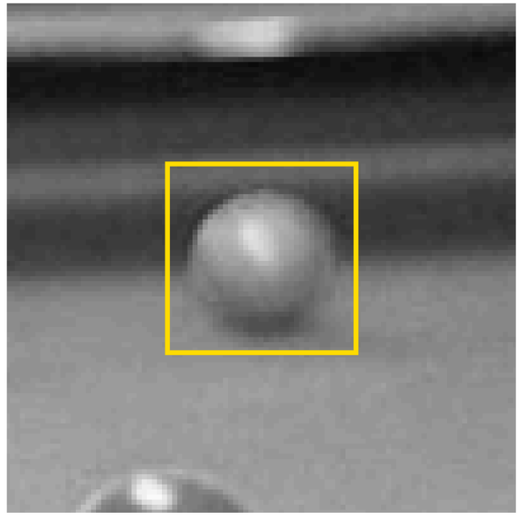
Filter the image with K_σ, then measure the strongest response inside each fixed yellow window:

$$R_\sigma = K_\sigma * I, \qquad m_W(\sigma) = \max_{(x,y)\in W} |R_\sigma(x,y)|.$$

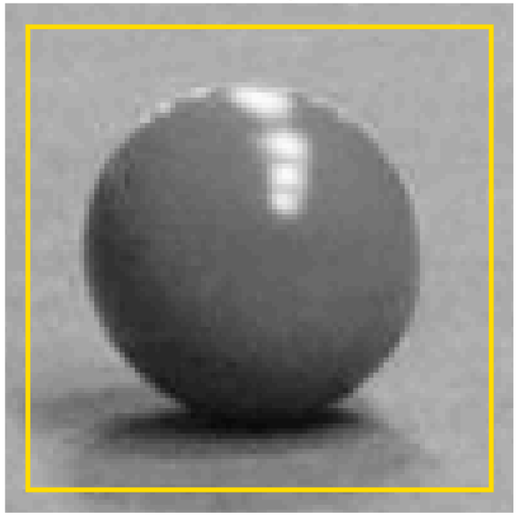
Full image: crop locations



Back: small ball



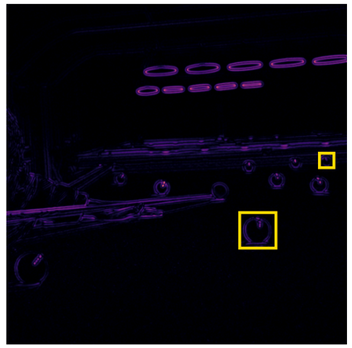
Front: large ball



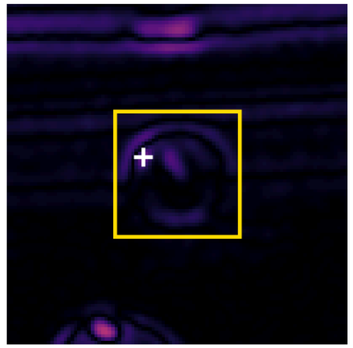
Colored boxes locate the crops. The yellow measurement windows stay fixed as σ changes.

Filter Response as the LOG Kernel Grows

Entire image response



Back: small ball

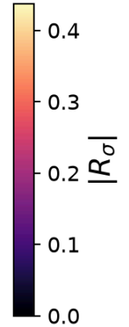


Max. $|R_\sigma| = 0.135$

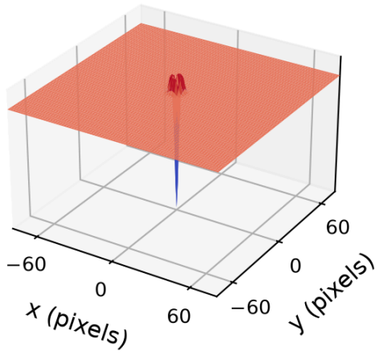
Front: large ball

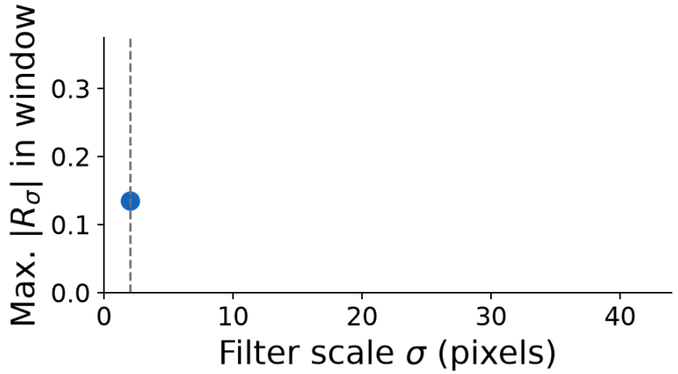


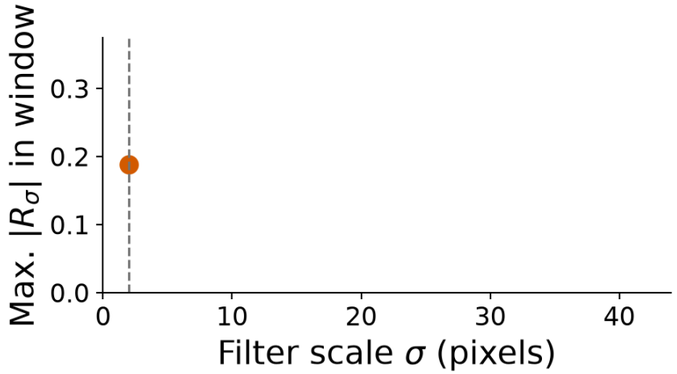
Max. $|R_\sigma| = 0.189$



LOG kernel: $\sigma = 2$



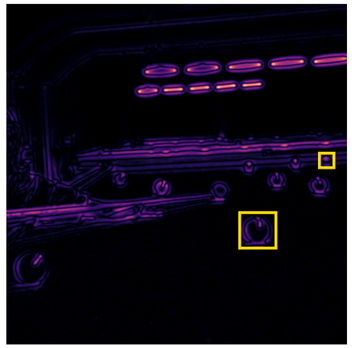




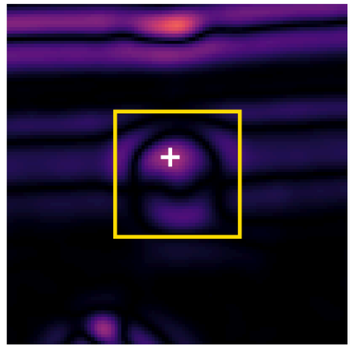
White crosses mark the window maxima. Small filters also respond to highlights.

Filter Response as the LOG Kernel Grows

Entire image response



Back: small ball

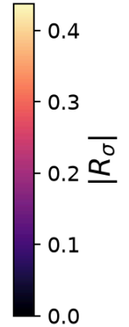


Max. $|R_\sigma| = 0.208$

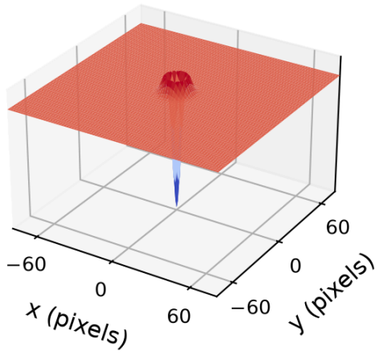
Front: large ball

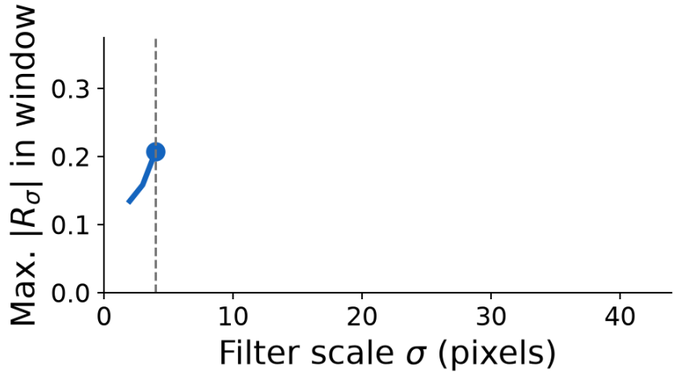


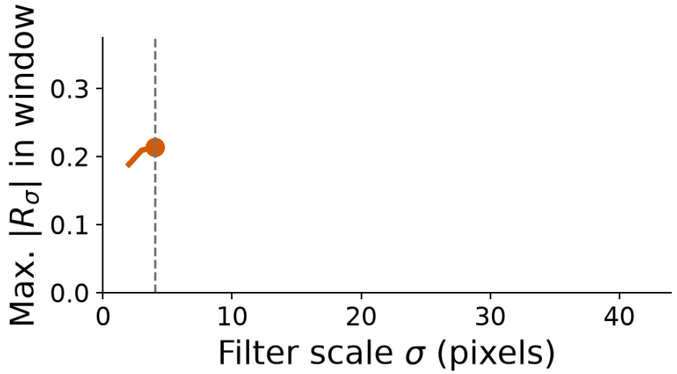
Max. $|R_\sigma| = 0.214$



LOG kernel: $\sigma = 4$



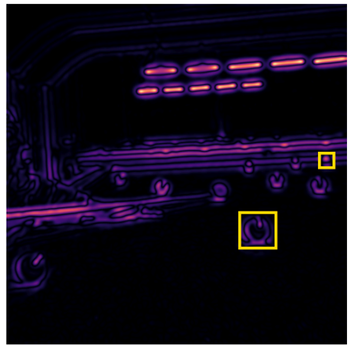




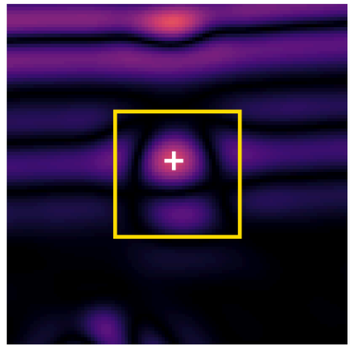
Both windows respond to fine detail at this small filter scale.

Filter Response as the LOG Kernel Grows

Entire image response



Back: small ball

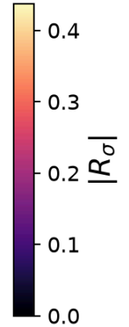


Max. $|R_\sigma| = 0.255$

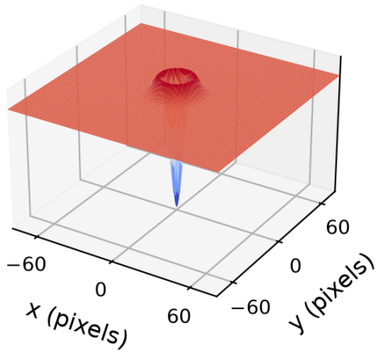
Front: large ball

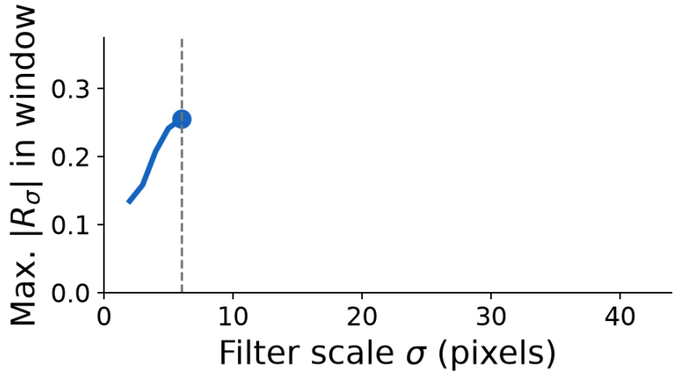


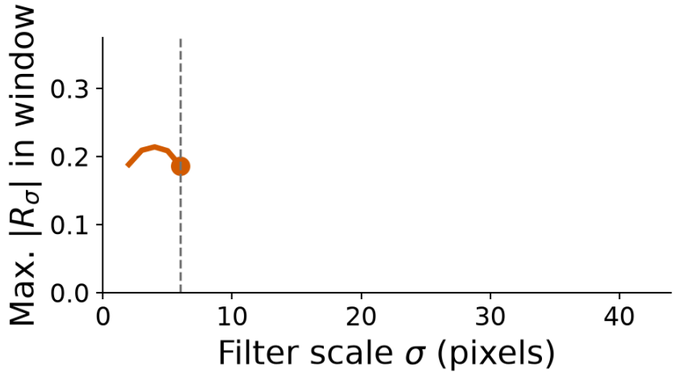
Max. $|R_\sigma| = 0.186$



LOG kernel: $\sigma = 6$



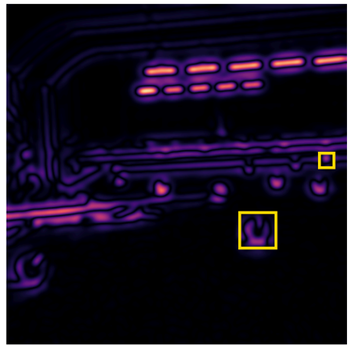




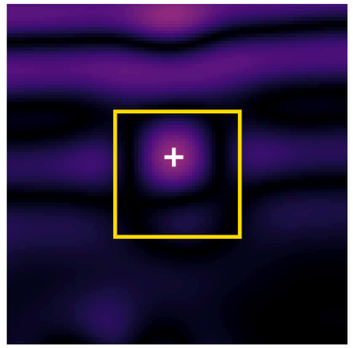
The small back ball reaches its **strongest response** at $\sigma = 6$.

Filter Response as the LOG Kernel Grows

Entire image response

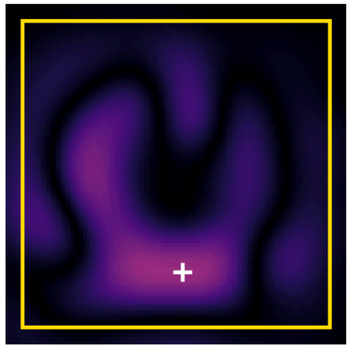


Back: small ball

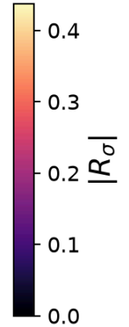


Max. $|R_\sigma| = 0.188$

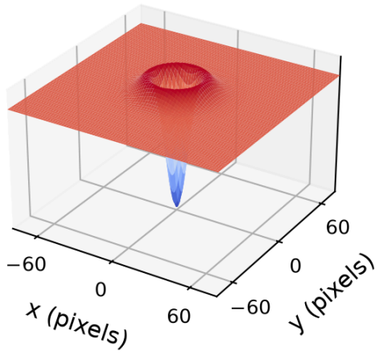
Front: large ball

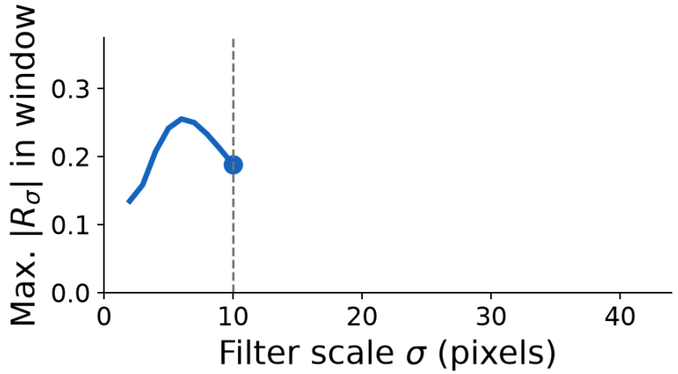


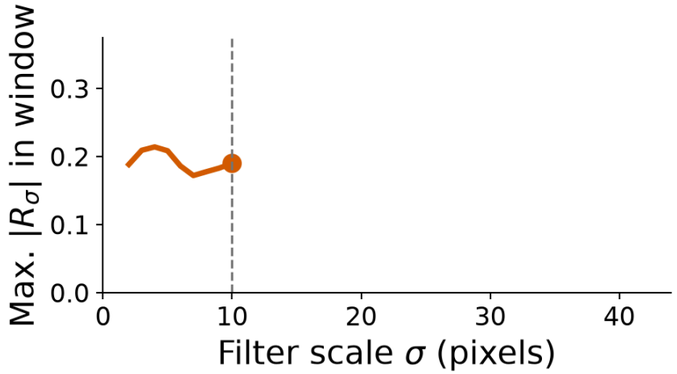
Max. $|R_\sigma| = 0.190$



LOG kernel: $\sigma = 10$



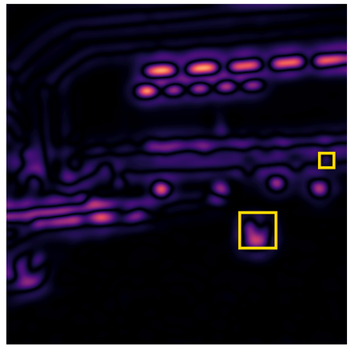




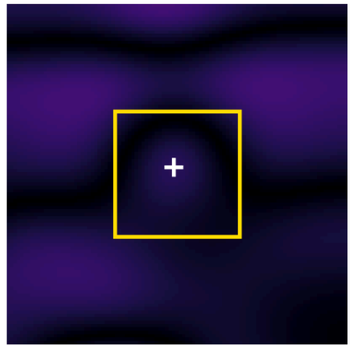
The back-ball response decreases as the filter grows beyond its size.

Filter Response as the LOG Kernel Grows

Entire image response

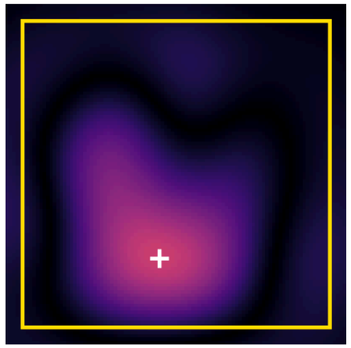


Back: small ball

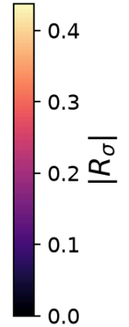


Max. $|R_\sigma| = 0.070$

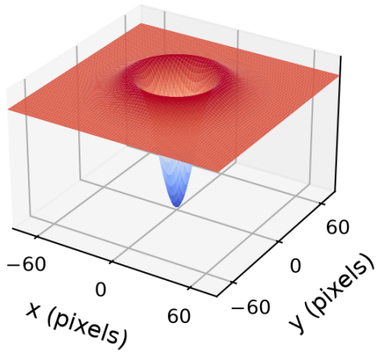
Front: large ball

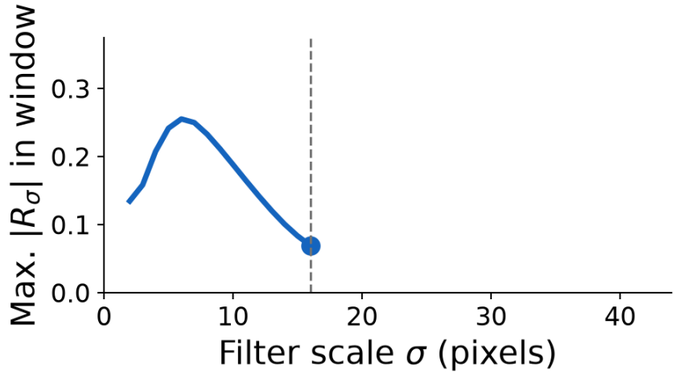


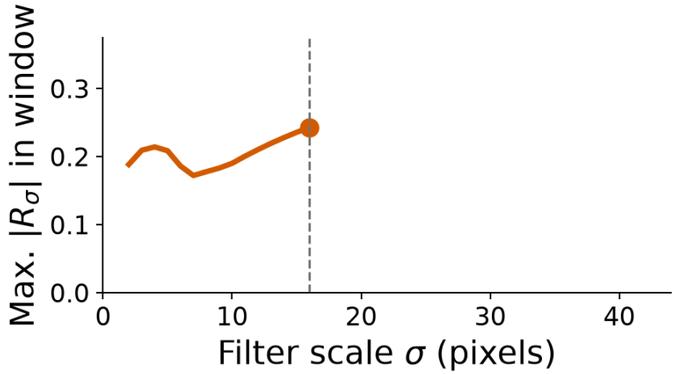
Max. $|R_\sigma| = 0.243$



LOG kernel: $\sigma = 16$



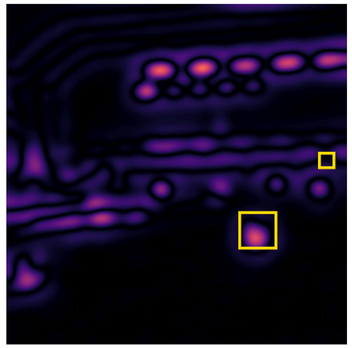




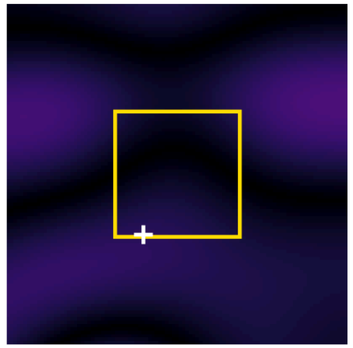
The larger filter now responds more strongly to the front ball.

Filter Response as the LOG Kernel Grows

Entire image response

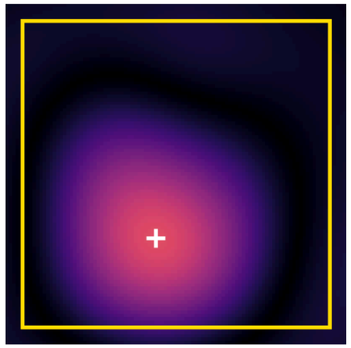


Back: small ball

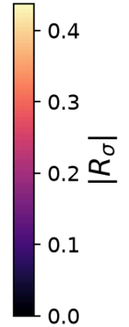


Max. $|R_\sigma| = 0.070$

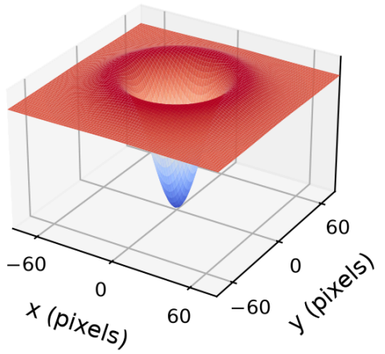
Front: large ball

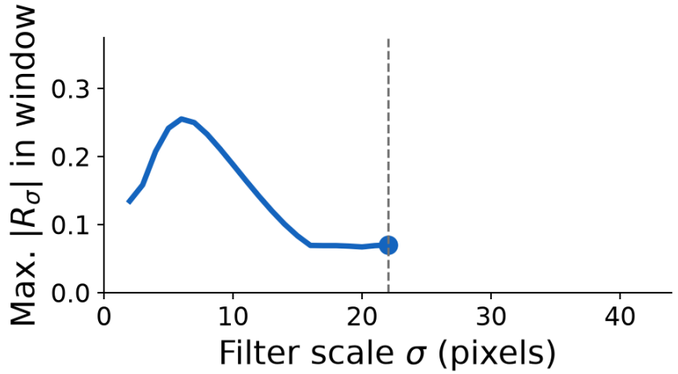


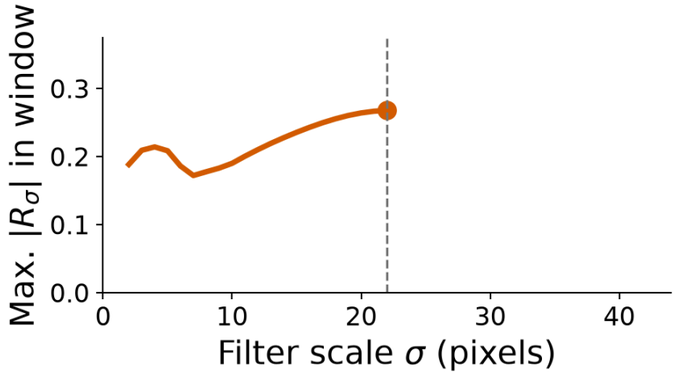
Max. $|R_\sigma| = 0.268$



LOG kernel: $\sigma = 22$



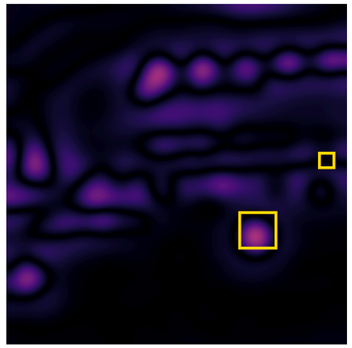




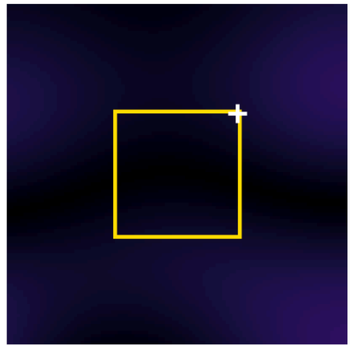
The large front ball reaches its **strongest response** at $\sigma = 22$.

Filter Response as the LOG Kernel Grows

Entire image response

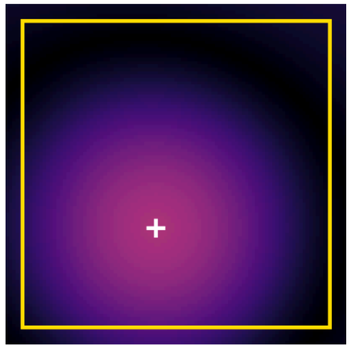


Back: small ball

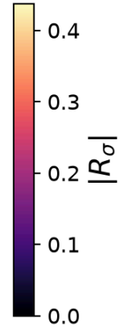


Max. $|R_\sigma| = 0.040$

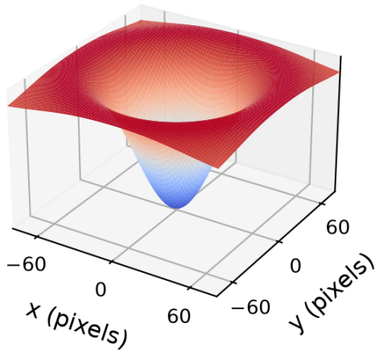
Front: large ball

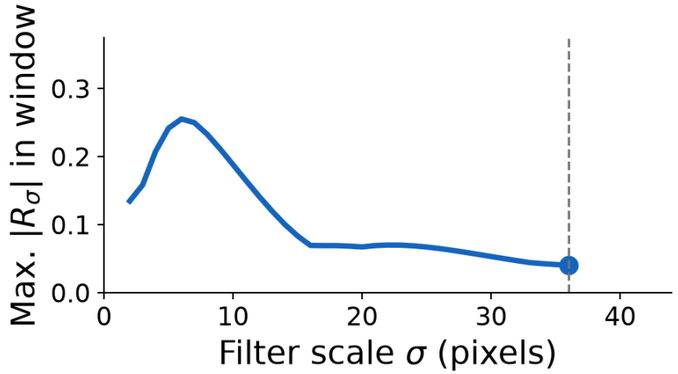


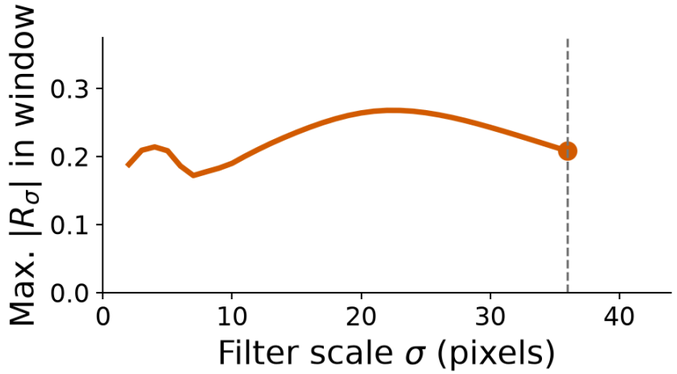
Max. $|R_\sigma| = 0.208$



LOG kernel: $\sigma = 36$

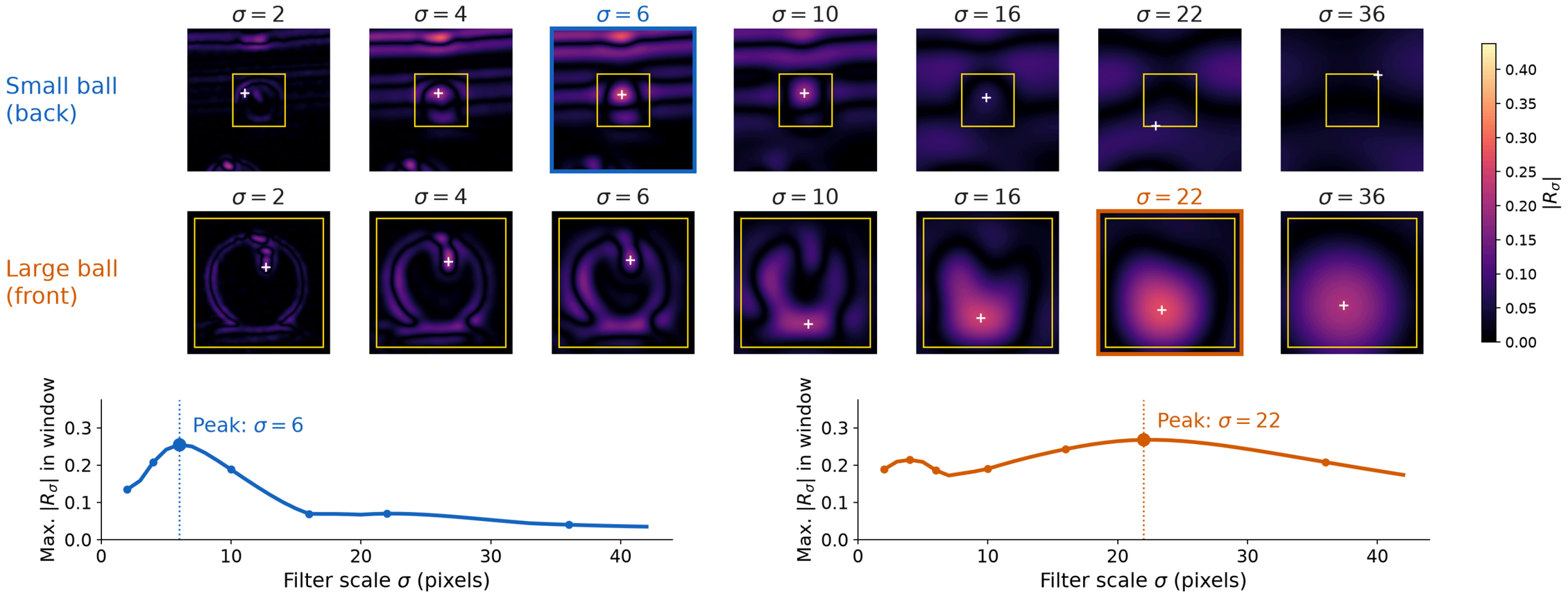






Both responses decrease once the filter grows beyond their selected scales.

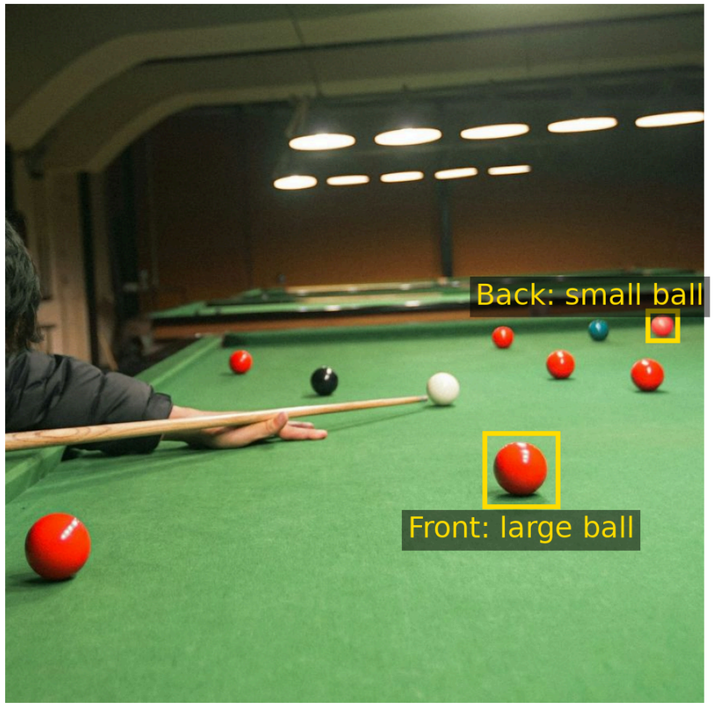
Small and Large Balls Select Different Scales

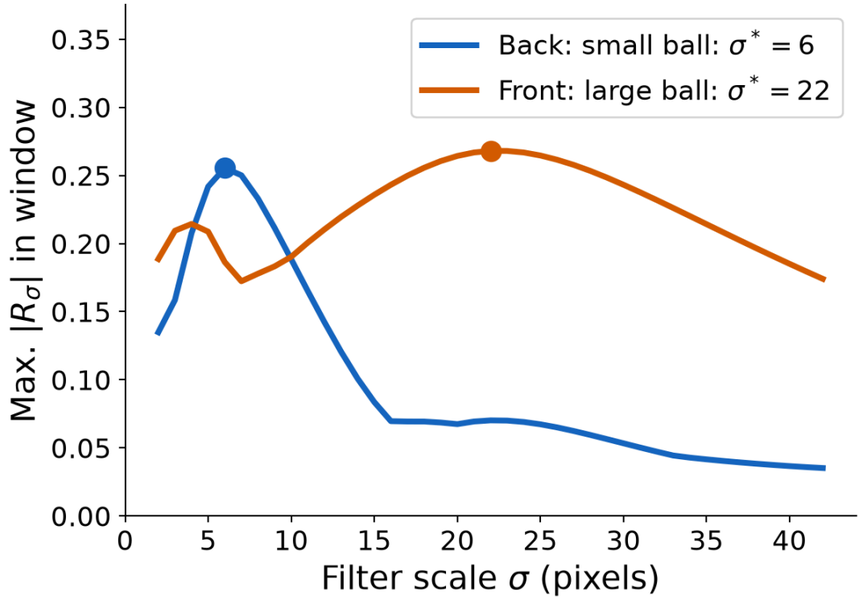


The small back ball peaks at a smaller filter scale. The large front ball peaks at a larger scale.

Automatic Scale Selection

Select each window's strongest scale: $\sigma_W^* = \arg\max_\sigma m_W(\sigma)$.





A single filter size cannot fit both balls. **The peak of each response curve selects its scale.**

Fixed Filter, Changing Image Size

- Keep $\sigma_0 = 4$ pixels and downsample the image by a factor d.
- The filter then covers an effective scale $\sigma_{\text{eff}} = d\sigma_0$ in the original image.

d = 1: 868 × 868 pixels



d = 2: 434 × 434 pixels



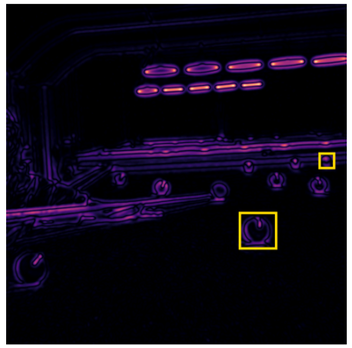
d = 4: 217 × 217 pixels



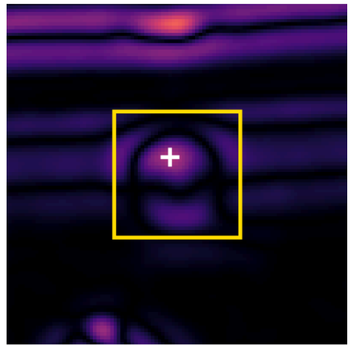
The windows follow the same balls as the image shrinks. Close-ups enlarge those regions for inspection.

Original Image, Fixed Filter

Full response: 868 × 868 pixels



Back: small ball

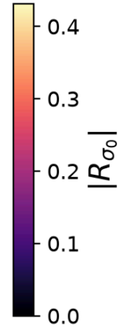


Max. $|R_{\sigma_0}| = 0.208$

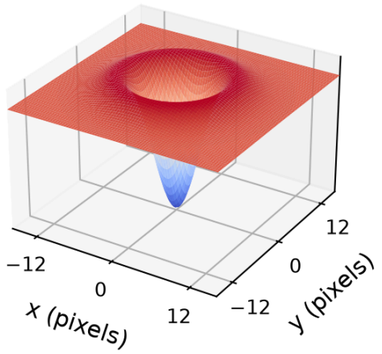
Front: large ball

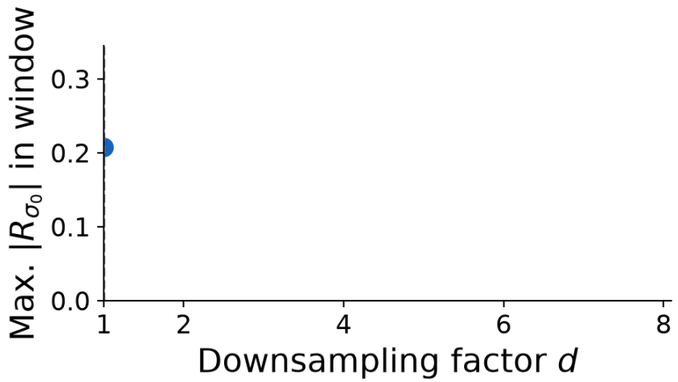


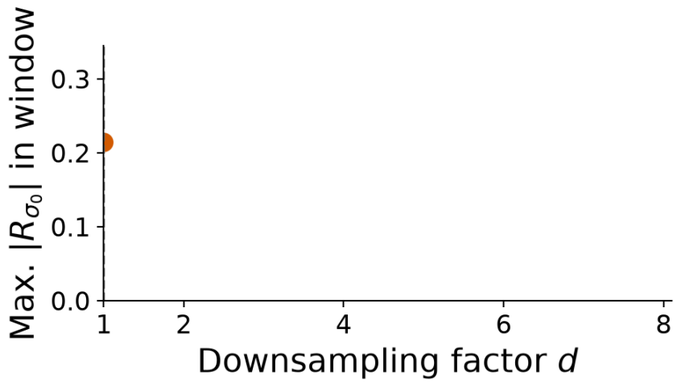
Max. $|R_{\sigma_0}| = 0.214$



LOG kernel: $\sigma = 4$



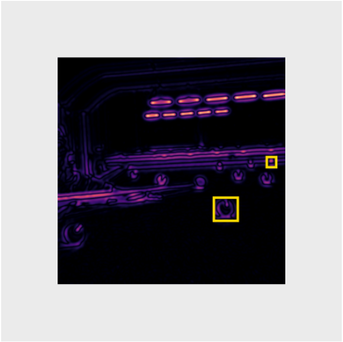




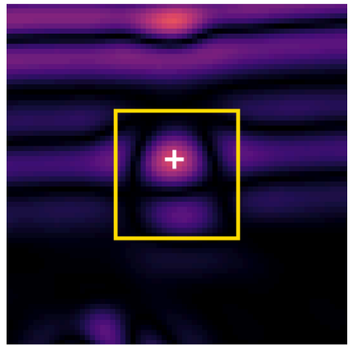
At full resolution, the fixed filter is small relative to both balls.

Smaller Image, Fixed Filter

Full response: 579 × 579 pixels

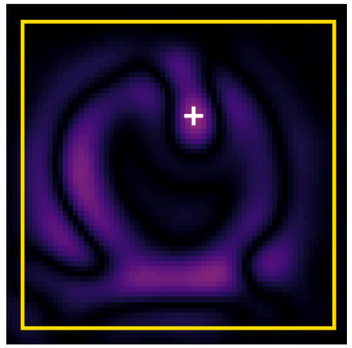


Back: small ball

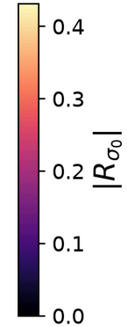


Max. $|R_{\sigma_0}| = 0.252$

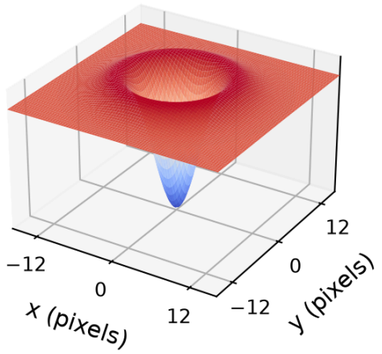
Front: large ball

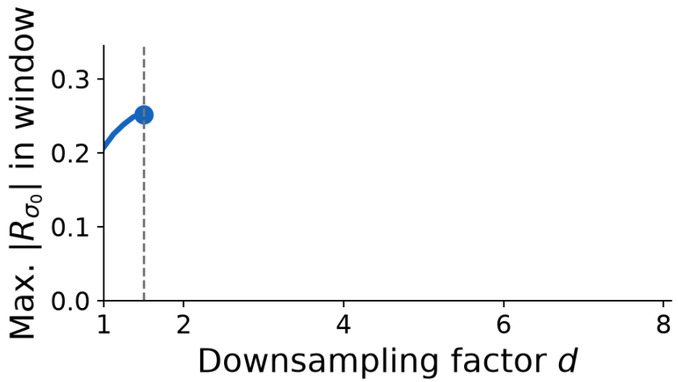


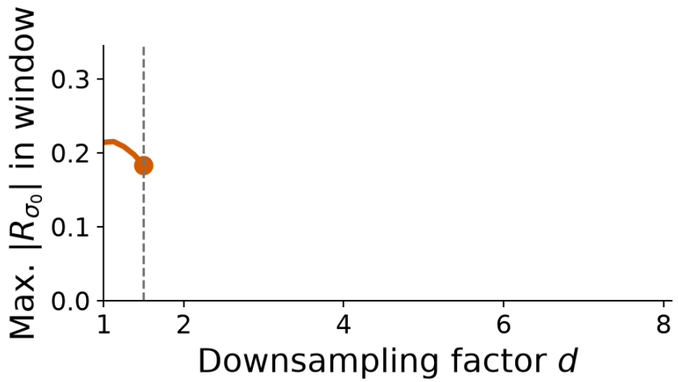
Max. $|R_{\sigma_0}| = 0.183$



LOG kernel: $\sigma = 4$



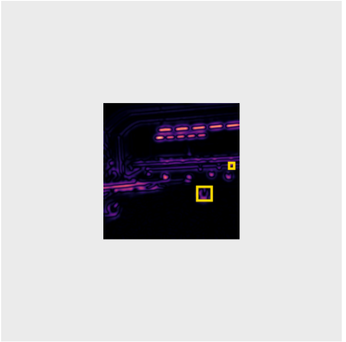




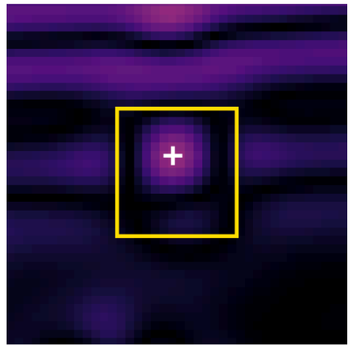
The small back ball reaches its peak near $d = 1.5$, with the filter still fixed at $\sigma_0 = 4$.

Smaller Image, Fixed Filter

Full response: 347 × 347 pixels

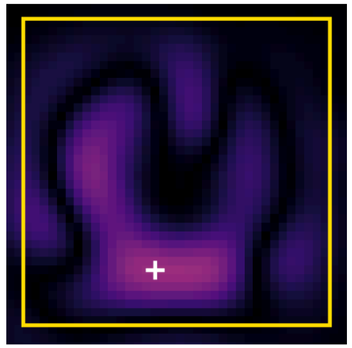


Back: small ball

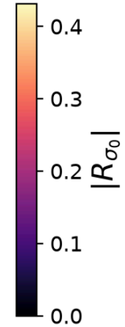


Max. $|R_{\sigma_0}| = 0.186$

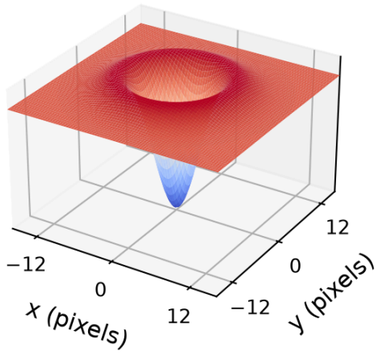
Front: large ball

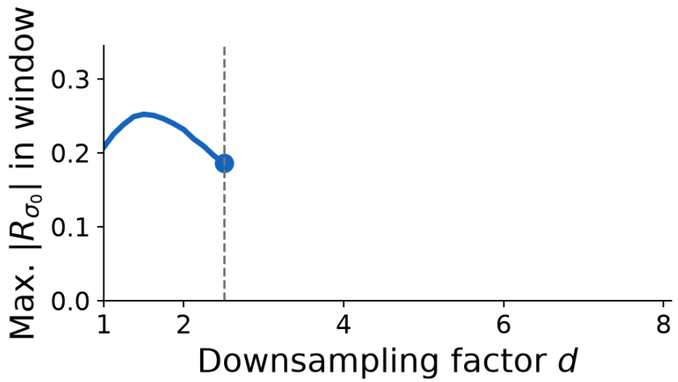


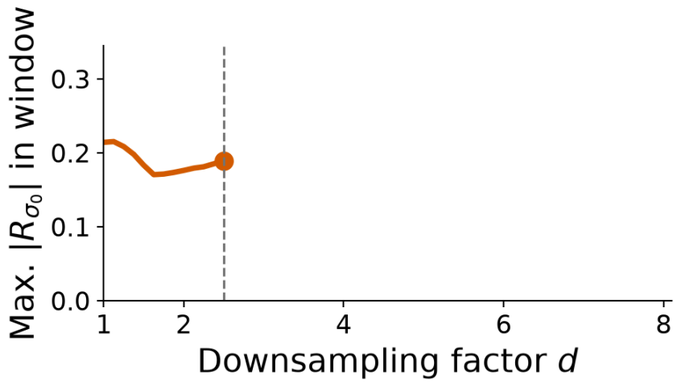
Max. $|R_{\sigma_0}| = 0.189$



LOG kernel: $\sigma = 4$







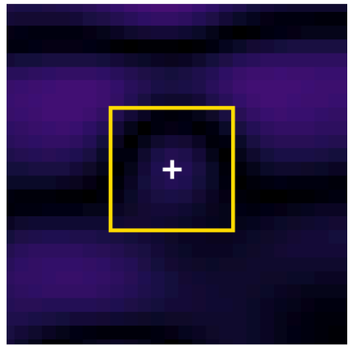
Further reduction makes the back ball too small for this filter.

Smaller Image, Fixed Filter

Full response: 217 × 217 pixels

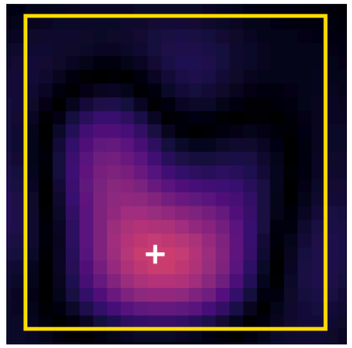


Back: small ball

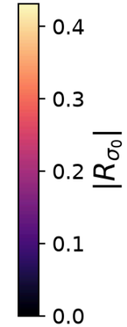


Max. $|R_{\sigma_0}| = 0.069$

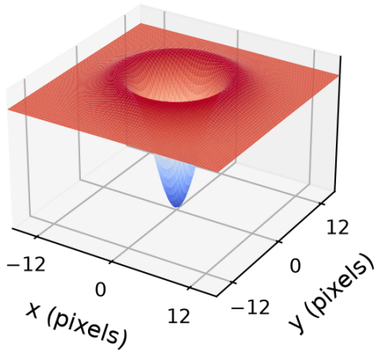
Front: large ball

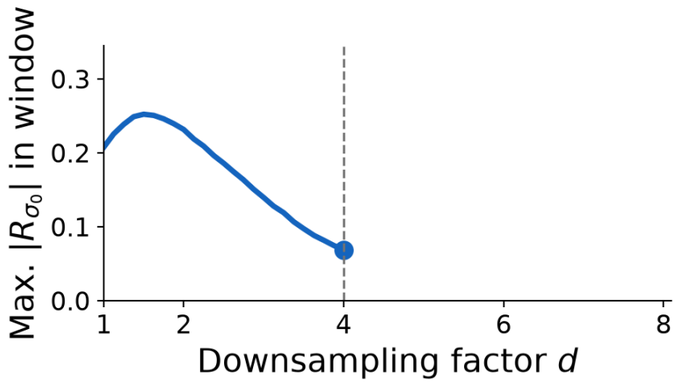


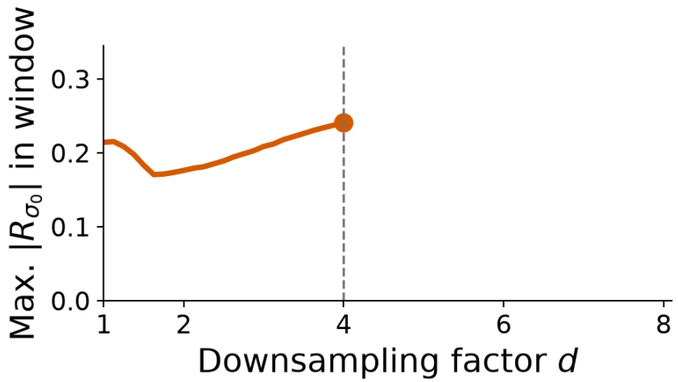
Max. $|R_{\sigma_0}| = 0.241$



LOG kernel: $\sigma = 4$



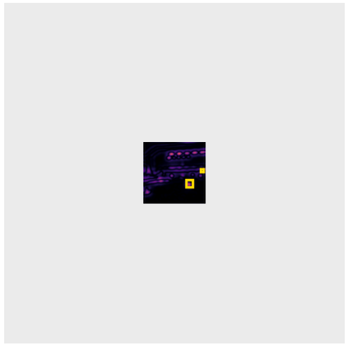




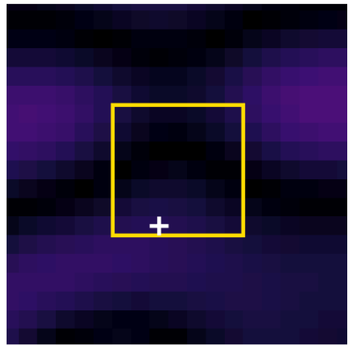
The front ball approaches the size that best matches the fixed filter.

Smaller Image, Fixed Filter

Full response: 158 × 158 pixels

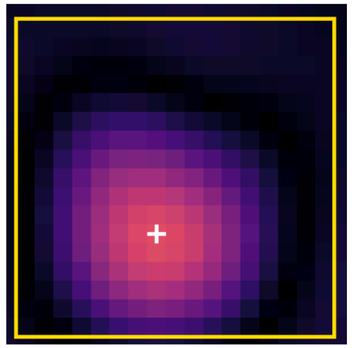


Back: small ball

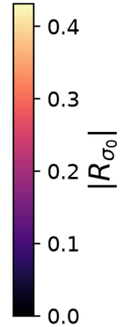


Max. $|R_{\sigma_0}| = 0.065$

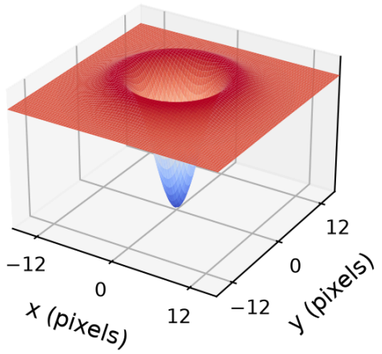
Front: large ball

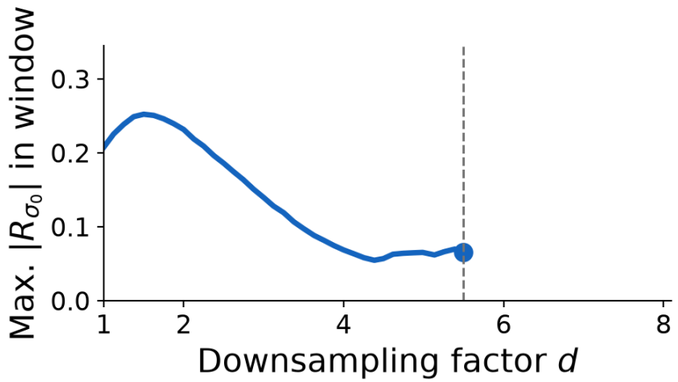


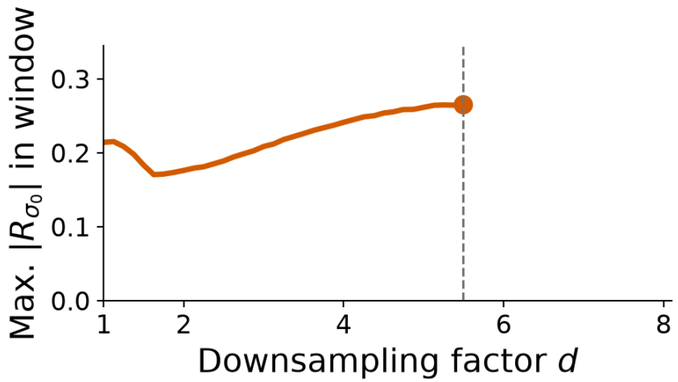
Max. $|R_{\sigma_0}| = 0.266$



LOG kernel: $\sigma = 4$



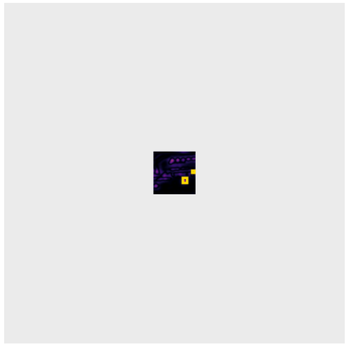




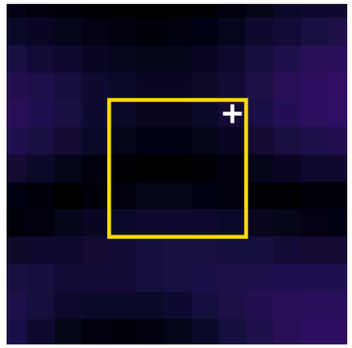
The large front ball reaches its peak near $d = 5.5$.

Smaller Image, Fixed Filter

Full response: 108 × 108 pixels

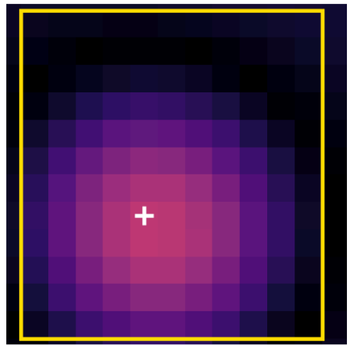


Back: small ball

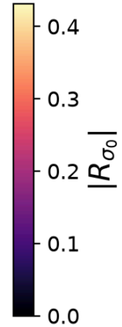


Max. $|R_{\sigma_0}| = 0.044$

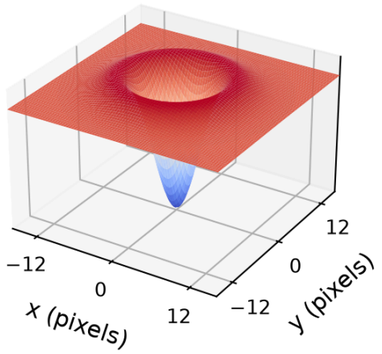
Front: large ball

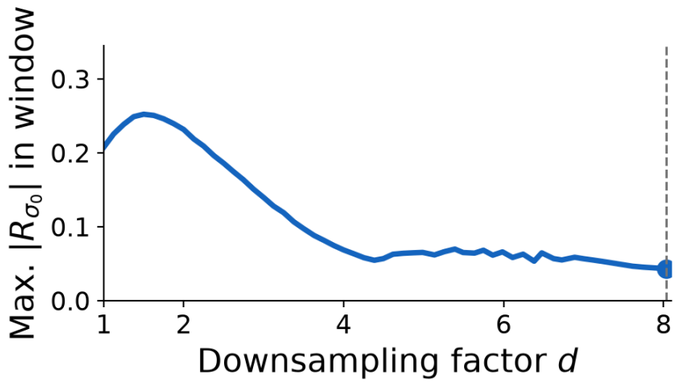


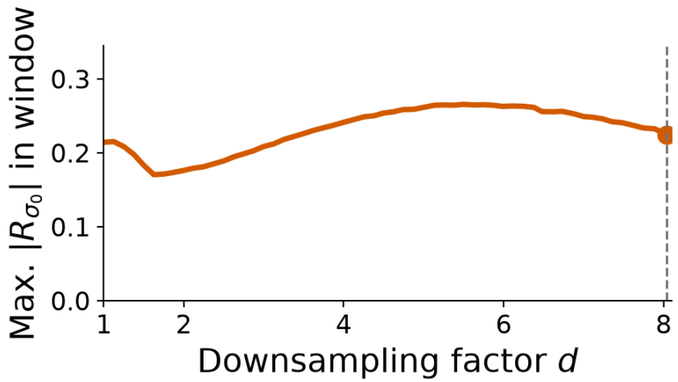
Max. $|R_{\sigma_0}| = 0.224$



LOG kernel: $\sigma = 4$



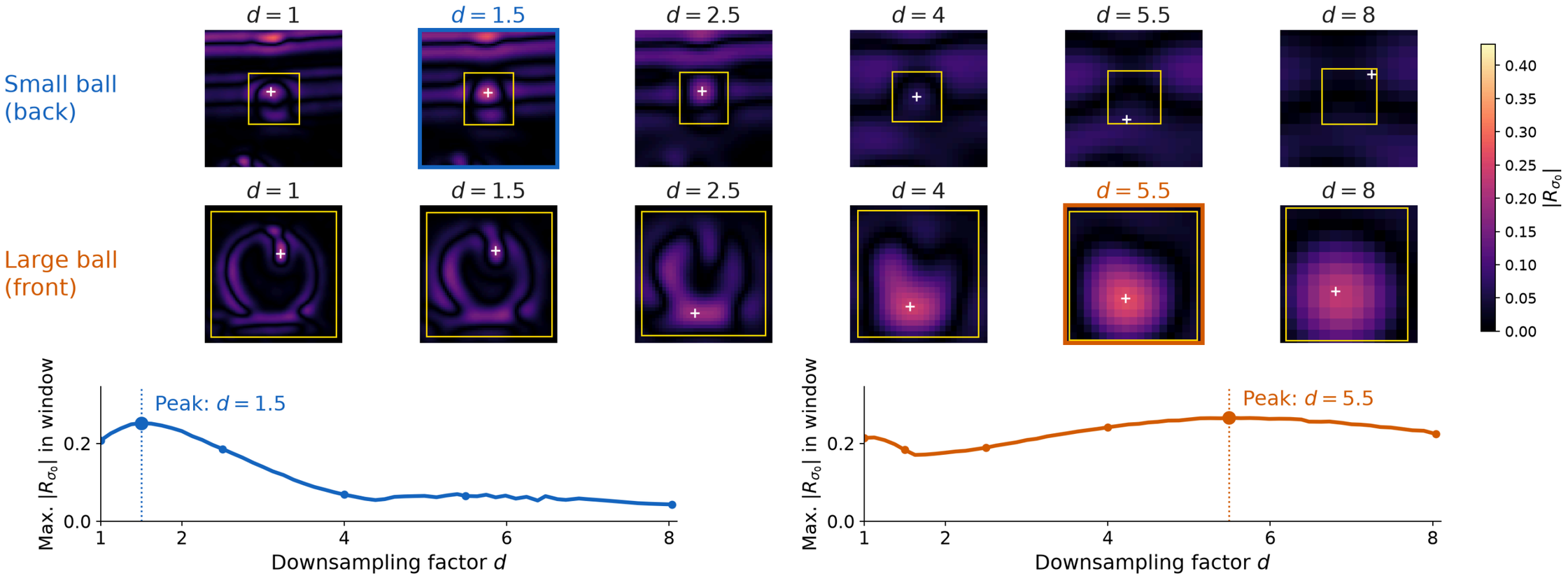




Both balls have shrunk past their strongest-response scales.

Different Balls Peak at Different Image Sizes

The filter stays fixed. Compare both balls across the downsampling factors d.

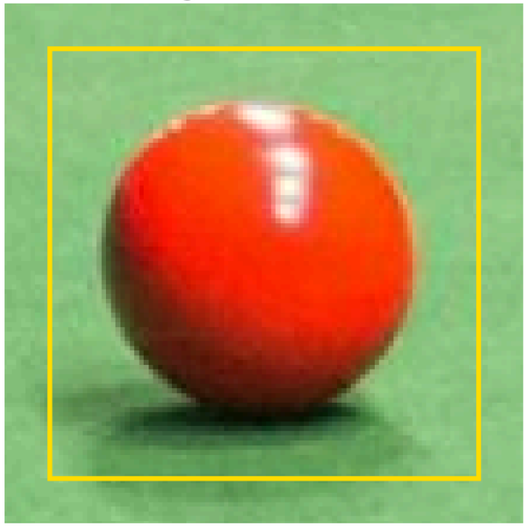


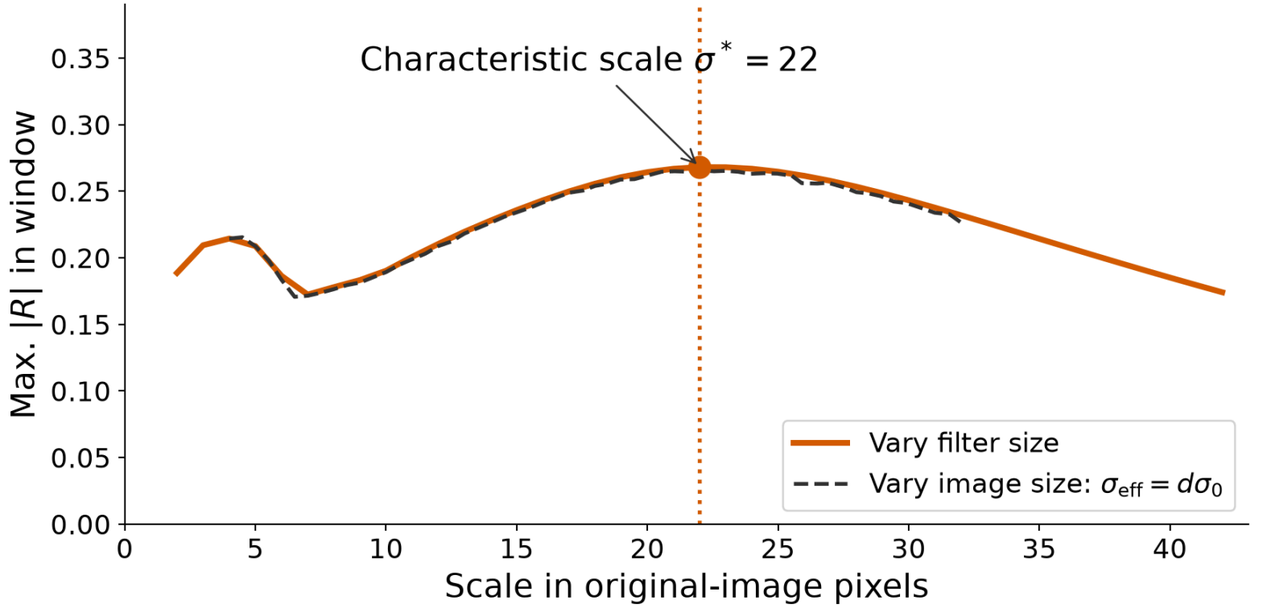
The larger ball needs more downsampling to match the same filter.

Characteristic Scale

The **characteristic scale** is the scale that produces the peak filter response.

Large front ball





Growing the filter or shrinking the image gives approximately the **same characteristic scale**.

Multi-Scale Detection

Multi-Scale Detection Algorithm

- For each filter size (or level of Gaussian pyramid)
  - compute feature response (e.g. Harris, Laplacian)
- For each filter size (or level of Gaussian pyramid)
  - if local maximum and cross-scale
    - save scale and location of feature (x, y, s)

Example: Multi-Scale Harris Corner Detection

1. Compute a comparable feature response $R(x, y, \sigma)$ at each Gaussian scale.
2. Find local response maxima within each image plane.
3. Compare responses across scales; keep strong peaks and their (x, y, σ).

$$\text{Image} \longrightarrow \{R_{\sigma_1}, R_{\sigma_2}, \ldots\} \longrightarrow \text{spatial peaks} \longrightarrow \text{scale selection}$$

We will use **Harris cornerness** and select the **strongest tested scale at each location**.

Example: Corners at Different Scales

```
IMAGE_PATH = "assets/07_features/mines_tower.jpg"
image = cv2.imread(IMAGE_PATH, cv2.IMREAD_GRAYSCALE)
image = image.astype(np.float32) / 255.0
scales = np.geomspace(1.0, 20.0, 19)
```



Small details and broad junctions need different analysis neighborhoods.

Harris Responses That Can Be Compared Across Scales

Use $I_x = \sigma\, \partial_x (G_\sigma * I)$ and $I_y = \sigma\, \partial_y (G_\sigma * I)$; average their products over $G_{1.5\sigma}$.

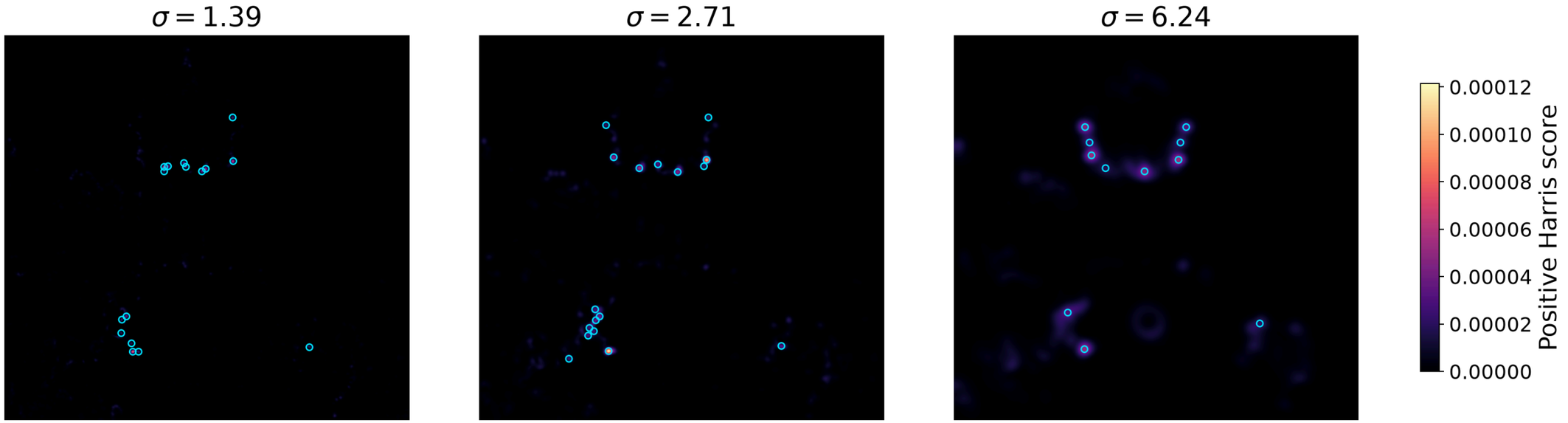
```python
def harris_response(image, sigma):
    smooth = cv2.GaussianBlur(image, (0, 0), float(sigma))
    Ix = cv2.Sobel(smooth, cv2.CV_32F, 1, 0, ksize=3, scale=sigma/8)
    Iy = cv2.Sobel(smooth, cv2.CV_32F, 0, 1, ksize=3, scale=sigma/8)
    A = cv2.GaussianBlur(Ix*Ix, (0, 0), float(1.5*sigma))
    B = cv2.GaussianBlur(Ix*Iy, (0, 0), float(1.5*sigma))
    C = cv2.GaussianBlur(Iy*Iy, (0, 0), float(1.5*sigma))
    return A*C - B*B - 0.04*(A + C)**2
```

$$M_\sigma = \begin{bmatrix} A & B \\ B & C \end{bmatrix}, \qquad R_\sigma = \det(M_\sigma) - 0.04\ \mathrm{tr}(M_\sigma)^2.$$

Both smoothing and the averaging neighborhood grow with σ.

Build the Stack and Find Spatial Peaks

```python
responses = np.stack([harris_response(image, s) for s in scales])
neighbors = np.ones((3, 3), np.uint8)
neighbors[1, 1] = 0                    # compare with the eight neighbors
neighbor_max = np.stack([cv2.dilate(r, neighbors) for r in responses])
spatial_peaks = responses > neighbor_max
```



Cyan marks are spatial peaks above the display threshold. **Their preferred scale is still unknown.**

Keep the Strongest Scale at Each Location

```python
best_scale = responses.argmax(axis=0)      # one scale index per pixel
levels = np.arange(len(scales))[:, None, None]
keep = spatial_peaks & (levels == best_scale)
keep &= responses > 0.15 * responses.max()
keep[[0, -1]] = False                      # require an interior scale peak
j, y, x = np.nonzero(keep)

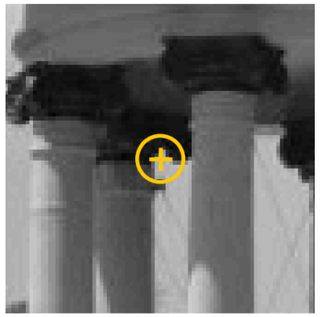
# remove detections too close to the image borders
margin = 4.5 * scales[j]                   # three integration standard deviations
h, w = image.shape
inside = (x > margin) & (x < w-margin) & (y > margin) & (y < h-margin)
j, y, x = j[inside], y[inside], x[inside]
selected_scales = scales[j]
```

$$\sigma^*(x, y) = \arg\max_{\sigma} R(x, y, \sigma).$$

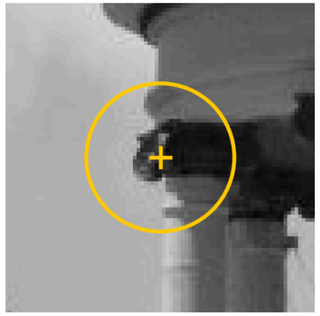
A surviving point is **a spatial peak at its winning scale**, with a strong positive score.

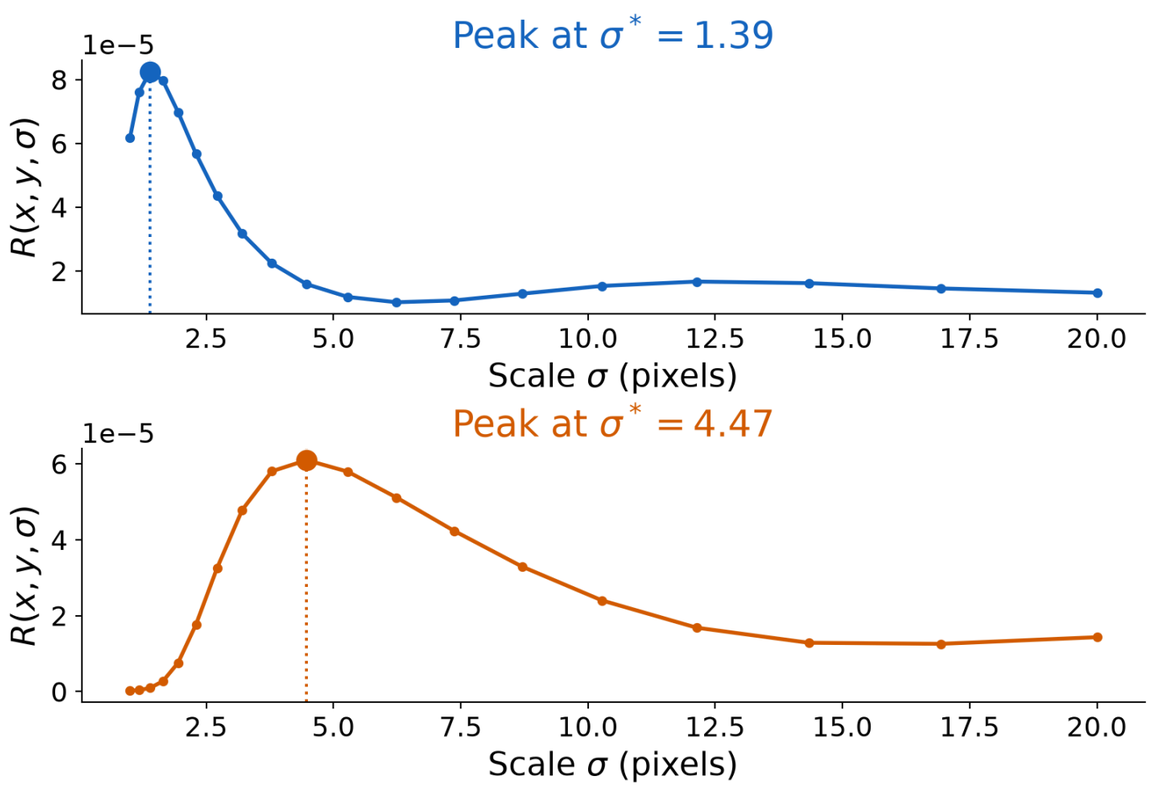
Two Corners, Two Winning Scales

$(x, y) = (249, 205),\ \sigma^* = 1.39$



$(x, y) = (213, 189),\ \sigma^* = 4.47$



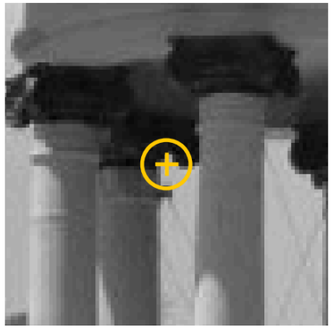


Each curve uses **one fixed pixel**. The yellow ring shows the averaging neighborhood ($3\sigma_I$).

Check the Location as Well as the Scale

The selected pixel must beat its **eight neighbors at the winning scale**.

Selected $\sigma^* = 1.39$

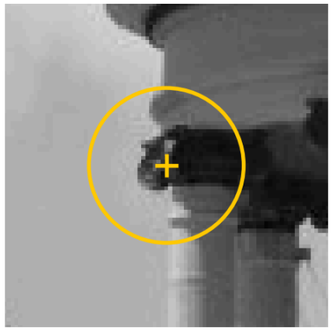


$\sigma = 1.00$		
3.390	5.943	5.809
3.858	6.164	5.713
3.116	4.385	3.807

$\sigma = 1.39$ (winner)		
5.205	6.935	6.708
6.373	8.246	7.869
6.356	7.850	7.336

$\sigma = 1.95$		
4.822	5.680	5.730
5.949	6.975	7.010
6.422	7.508	7.566

Selected $\sigma^* = 4.47$



$\sigma = 3.21$		
4.856	4.505	4.013
5.197	4.782	4.220
5.264	4.812	4.214

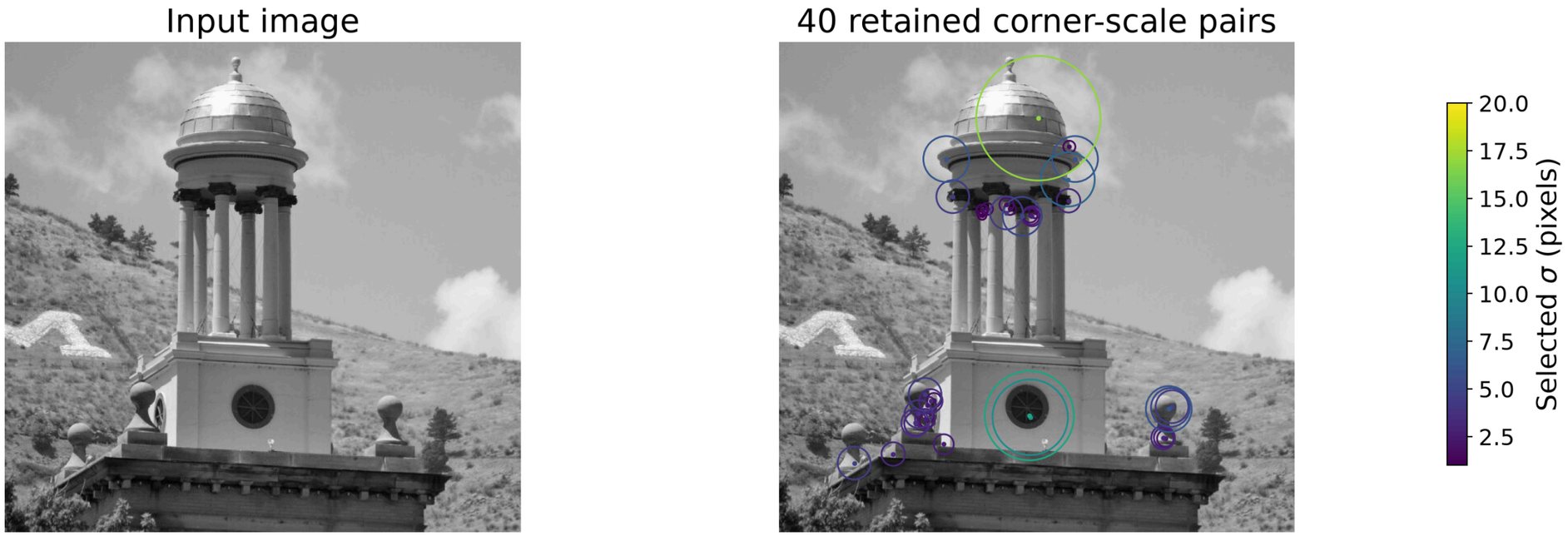
$\sigma = 4.47$ (winner)		
6.061	6.093	5.969
6.088	6.095	5.947
5.930	5.918	5.754

$\sigma = 6.24$		
5.033	5.211	5.320
4.940	5.109	5.211
4.771	4.930	5.023

Numbers are Harris scores $\times 10^5$. Cyan boxes mark the **same pixel** in every response patch.

Output: Corner Locations and Their Analysis Scales

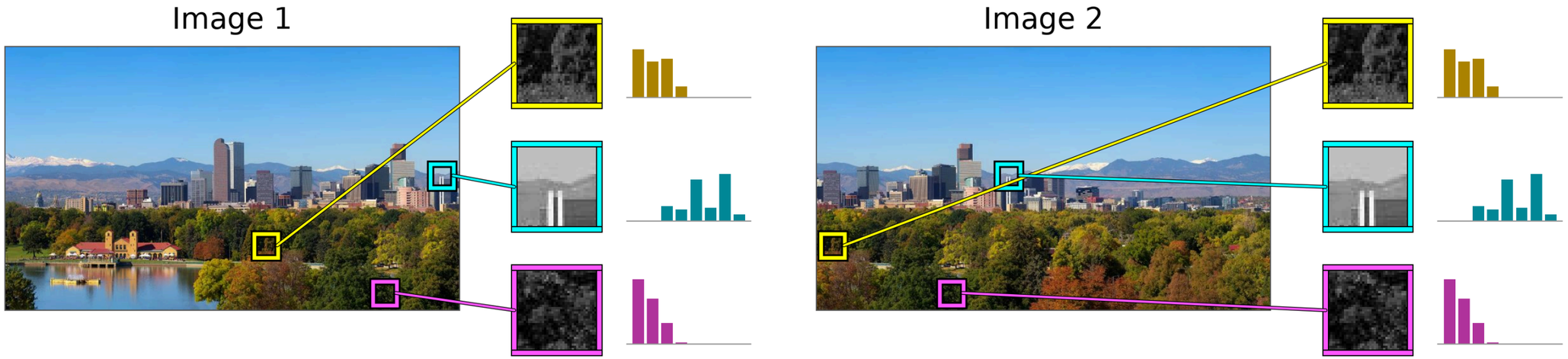
```
corners = np.column_stack([x, y, selected_scales])  # each row: (x, y, sigma)
```



- **Scale describes the analysis neighborhood, not an object boundary.**
- These are maxima over the tested scales; not every corner has a unique preferred scale (that is, some corners remain strong over a broad range of filter sizes, e.g., an ideal L-shaped corner looks similar as you zoom in or out).

Local Feature Matching: Main Components

We have locations and scales. Next, describe and match the features.



1. Detection

Find a set of distinctive features (aka, keypoints).

2. Description

Extract a feature descriptor around each interest point as a vector.

3. Matching

Compute distances between feature vectors to find correspondences.

Why Do We Need Feature Descriptors?

Detection tells us **where** a feature is. How do we recognize the same feature in another image?

Image 1

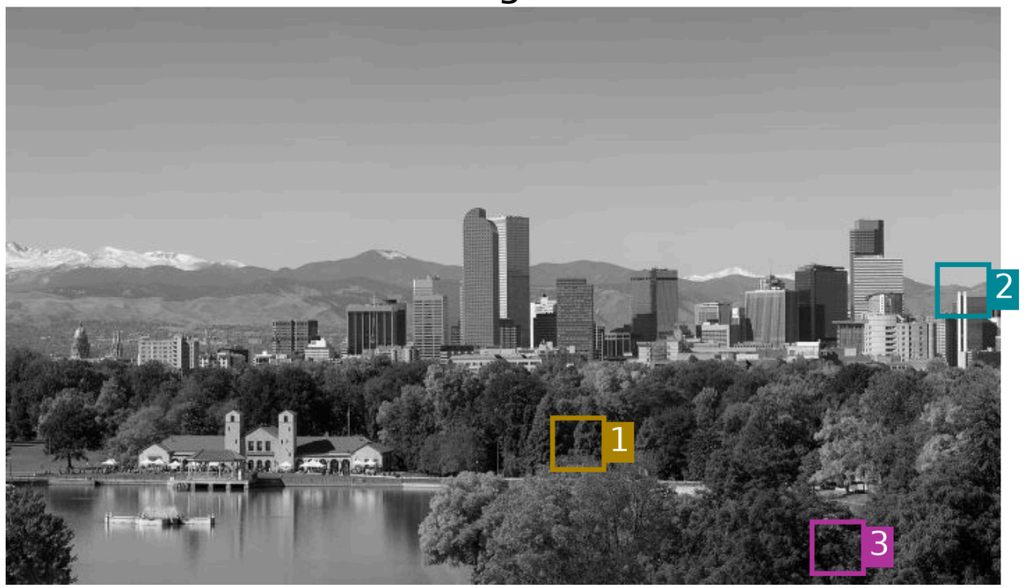


Image 2



A **descriptor** encodes the appearance around a keypoint as a vector. Comparing these vectors helps us find corresponding features.

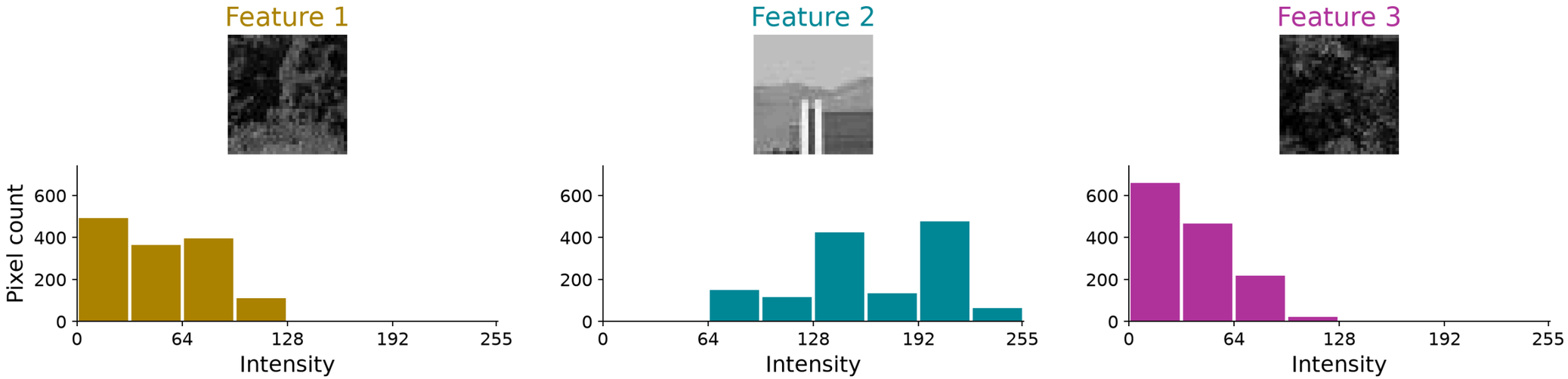
Possible Feature Descriptors

Templates

- Pixel intensities in a patch.
- Image gradients in a patch.

Histograms

- Intensity, color, or texture measurements.
- Gradient orientations (as used in SIFT).



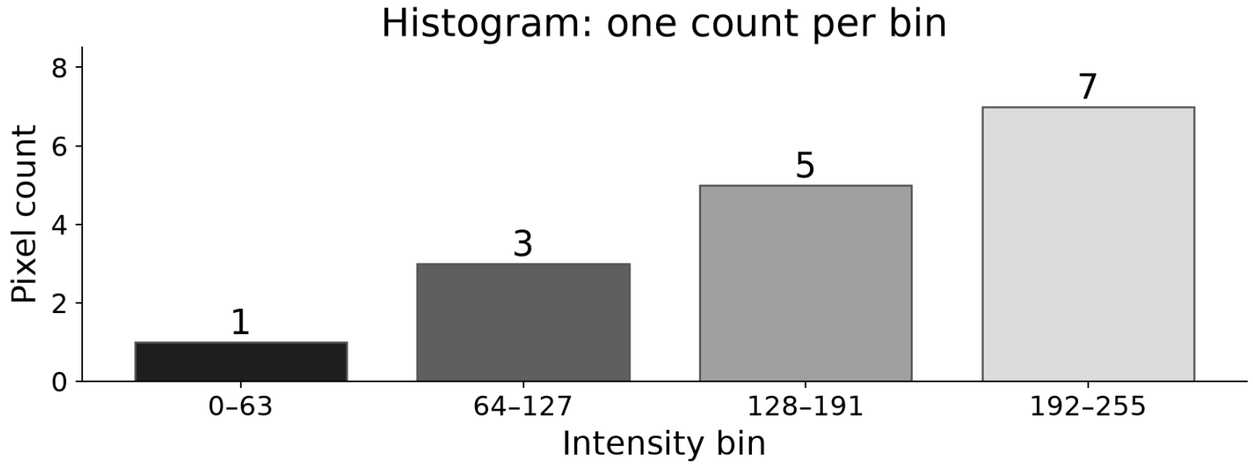
Example: an **8-bin intensity histogram** turns each patch into an 8-number descriptor.

Feature Descriptor: Histograms

A histogram divides values into **bins** and counts how many fall into each bin.

A 4 × 4 grayscale patch

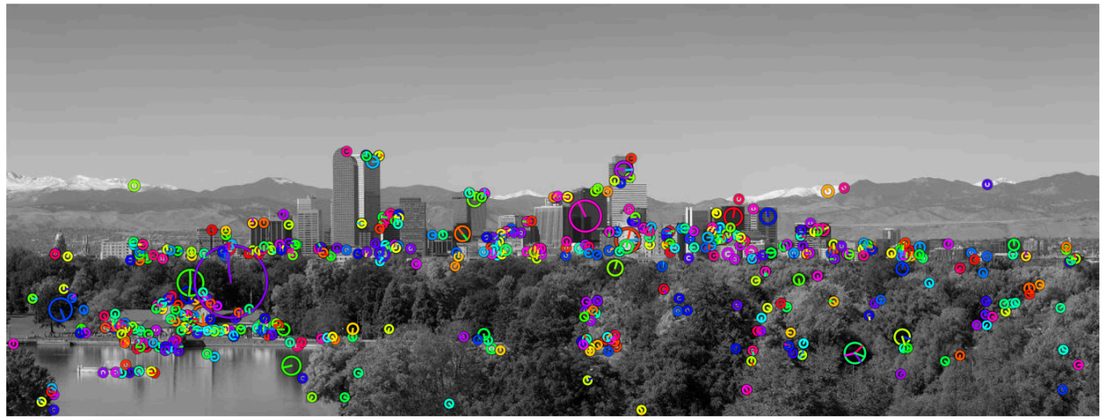
32	96	96	96
160	160	160	160
160	224	224	224
224	224	224	224



The histogram descriptor is $\mathbf{h} = [1,\ 3,\ 5,\ 7]$, with 16 pixels in total.

- **Global histogram:** use the whole image.
- **Local histogram:** use a patch around a keypoint.
- A histogram records how often values occur, but not their pixel positions within the patch.

SIFT: Scale-Invariant Feature Transform



David G. Lowe, "**Distinctive Image Features from Scale-Invariant Keypoints,**" *International Journal of Computer Vision*, 2004 (~83,000 citations). Link to original paper.

- Detect repeatable keypoints across image scales.
- Describe their neighborhoods in a consistent scale and orientation.
- Compare descriptors to find corresponding features across images.

$$\underbrace{(x, y, \sigma, \theta)}_{\text{keypoint frame}} \quad + \quad \underbrace{\mathbf{d} \in \mathbb{R}^{128}}_{\text{appearance descriptor}}$$

SIFT: Main Steps



SIFT Algorithm

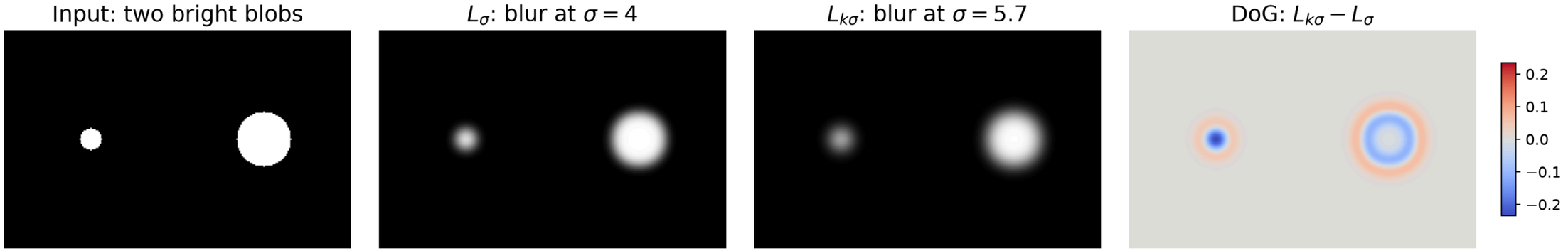
- Detect keypoints across scales
  - Find extrema of the Difference-of-Gaussians (DoG) response.
  - Refine their locations and scales. Reject weak and edge-like responses.
- Assign an orientation to each keypoint
  - Find dominant directions in the local image gradients.
- Construct a descriptor in the keypoint's coordinate frame
  - Account for its location, scale, and orientation.
  - Accumulate gradient orientations in a 4×4 array of local histograms.
- Normalize the descriptor
  - Normalize, limit large components, and normalize again.

DoG: Subtract Two Blurred Images

Blur the same image at σ and $k\sigma$, then subtract **pixel by pixel**:

$$L_\sigma = G_\sigma * I, \qquad D_\sigma = L_{k\sigma} - L_\sigma = (G_{k\sigma} - G_\sigma) * I.$$

```
dog_sigma, dog_k = 4.0, np.sqrt(2)
dog_L1 = cv2.GaussianBlur(dog_image, (0, 0), dog_sigma)
dog_L2 = cv2.GaussianBlur(dog_image, (0, 0), dog_k*dog_sigma)
dog_difference = dog_L2 - dog_L1
```

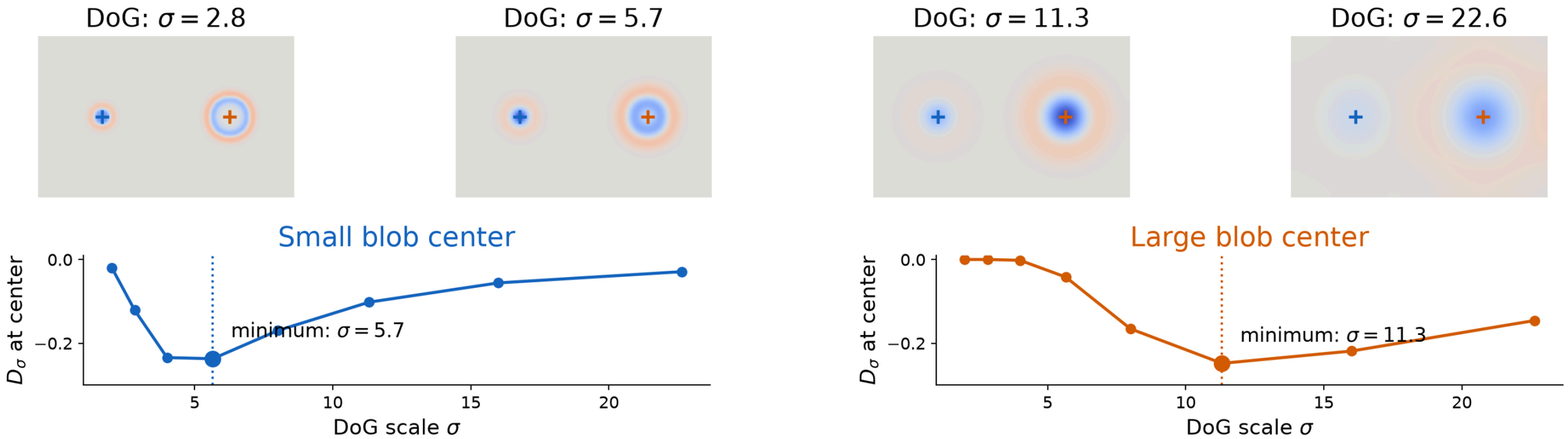


More blur lowers each bright peak: the DoG has a **negative center** (blue) and a positive surround (red). A uniform region gives approximately zero.

Search Across Filter Sizes

Repeat the subtraction at increasing scales. Here, all images stay at the same resolution.

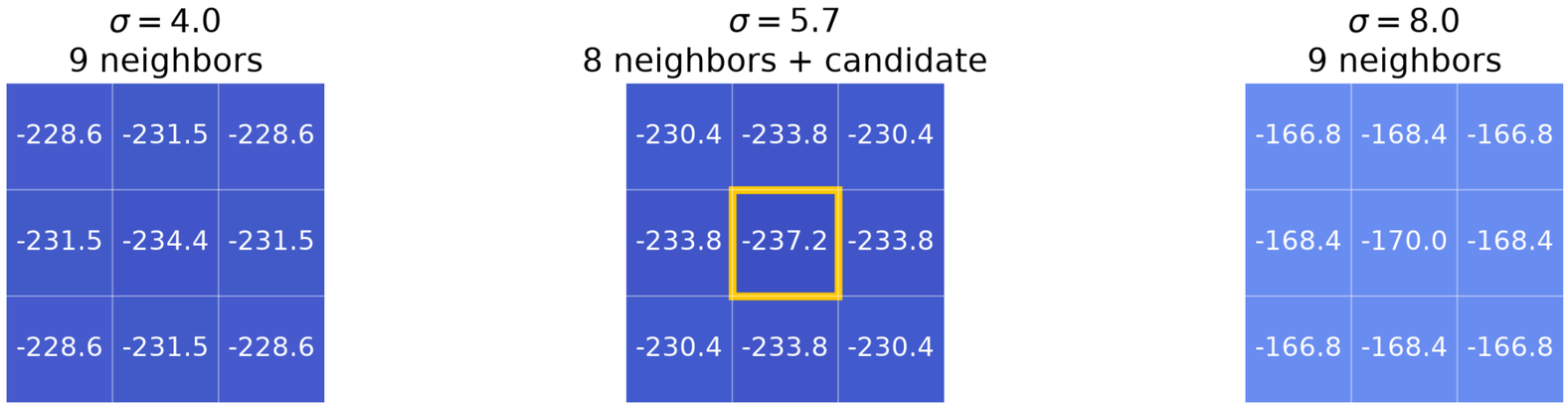
```
dog_scales = 2.0 * dog_k**np.arange(9)
dog_L = [cv2.GaussianBlur(dog_image, (0, 0), float(s))
    for s in dog_scales]
dog_D = np.diff(np.stack(dog_L), axis=0)  # D[j] = L[j+1] - L[j]
dog_response_scales = dog_scales[:-1]
```



The small blob reaches its strongest negative response at $\sigma \approx 5.7$; the large blob at $\sigma \approx 11.3$. **Each feature selects a different scale.**

Is It an Extremum in Both Position and Scale?

Compare a candidate with **8 neighbors at the same scale + 9 at the finer scale + 9 at the coarser scale.**



Numbers show $1000D_\sigma$. The boxed value is smaller than **all 26 neighbors**, so it survives as a DoG minimum. We also retain maxima.

```python
def dog_is_extremum(stack, j, y, x):
    cube = stack[j-1:j+2, y-1:y+2, x-1:x+2]
    neighbors = np.delete(cube.ravel(), 13)  # remove the central sample
    value = stack[j, y, x]
    return (value < neighbors.min()) or (value > neighbors.max())
```

Detect Keypoints by Scanning the Whole Stack

Search all locations and scales with a full neighborhood. Skip weak responses using a contrast threshold.

```
dog_threshold = 0.08                              # chosen for this example
interior = dog_D[1:-1, 1:-1, 1:-1]
candidates = np.argwhere(np.abs(interior) > dog_threshold) + 1

dog_detections = []
for j, y, x in candidates:
    if dog_is_extremum(dog_D, j, y, x):
        dog_detections.append((x, y, dog_response_scales[j]))
```



Each candidate now has **a position and a scale** (x, y, σ); circles show radius 2σ. SIFT then refines these estimates and rejects unstable edge-like candidates.

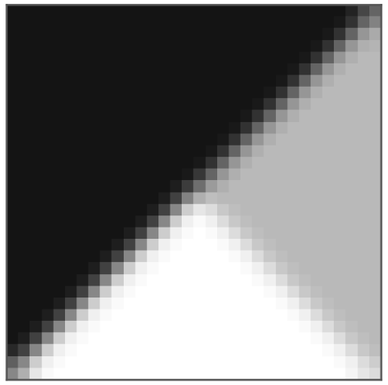
The Descriptor Starts with Local Gradients

At the detected scale, gradients describe the direction and strength of local brightness changes.

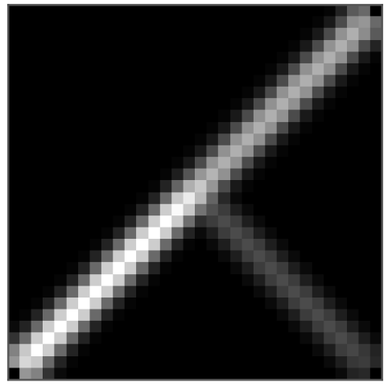
```python
def sf_gradients(patch):
    gx = cv2.Sobel(patch, cv2.CV_32F, 1, 0, ksize=3, scale=1/8)
    gy = cv2.Sobel(patch, cv2.CV_32F, 0, 1, ksize=3, scale=1/8)
    return cv2.cartToPolar(gx, gy, angleInDegrees=True)

sf_mag, sf_angle = sf_gradients(sf_patch)
```

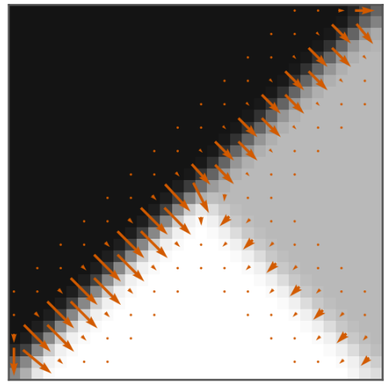
Input image patch



Gradient magnitude



Gradient arrows



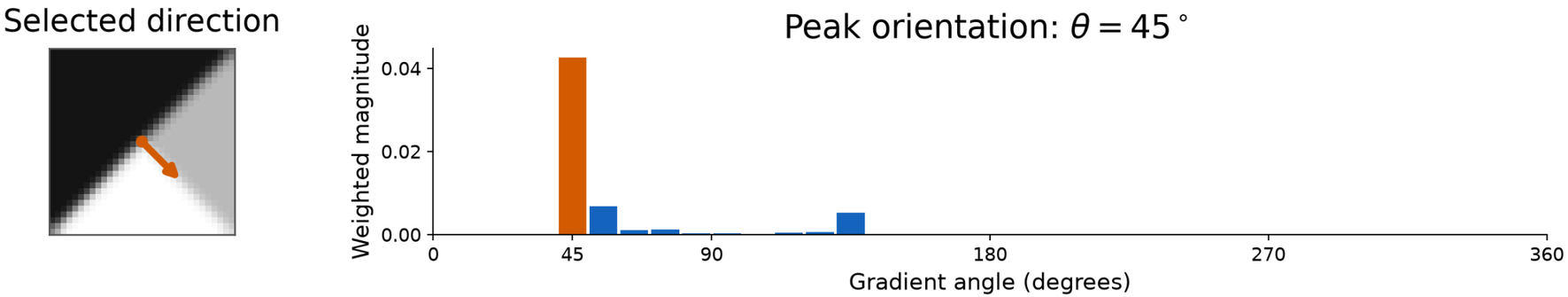
Arrows point toward brighter pixels. Longer arrows indicate stronger gradients. The main direction is approximately 45°.

Assign a Dominant Orientation

- Add gradient magnitudes to a **36-bin orientation histogram**.
- Use Gaussian weights for orientations so that the dominant orientation is mainly based on the keypoint's neighboring pixels.

```
def sf_orientation(patch, sigma):
    mag, angle = sf_gradients(patch)
    g = cv2.getGaussianKernel(len(patch), 1.5*sigma)
    bins = (angle // 10).astype(int) % 36
    hist = np.bincount(bins.ravel(),
                       weights=(mag*(g @ g.T)).ravel(), minlength=36)
    return 10*(hist.argmax() + 0.5), hist

sf_theta, sf_orientation_hist = sf_orientation(sf_patch, sigma=4)
```



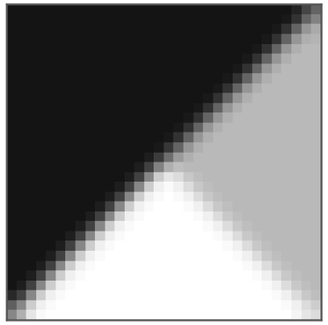
- The dominant orientation is colored orange in this example.
- Note that angles increase clockwise, and image coordinates have x pointing to the right and y pointing downward.

Use a Common Scale and Orientation

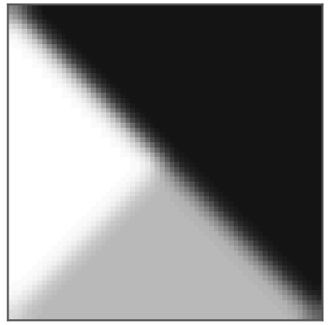
Sample a window proportional to keypoint scale, with its axes aligned to the selected orientation.

```python
def sf_canonical(patch, theta):
    n = len(patch)
    M = cv2.getRotationMatrix2D(((n-1)/2, (n-1)/2), theta, 1)
    aligned = cv2.warpAffine(patch, M, (n, n))
    width = int(n/np.sqrt(2))           # central region with valid samples
    start = (n-width)//2
    crop = aligned[start:start+width, start:start+width]
    return cv2.resize(crop, (16, 16), interpolation=cv2.INTER_AREA)
```

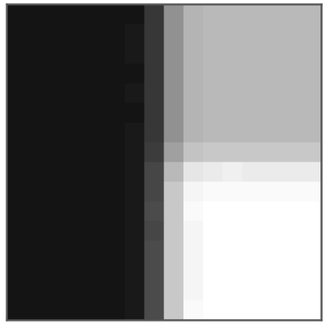
View A: $\theta = 45^\circ$
32 × 32 crop



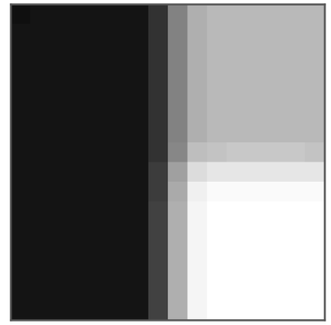
View B: $\theta = 135^\circ$
64 × 64 crop



A in keypoint frame
16 × 16 samples



B in keypoint frame
16 × 16 samples



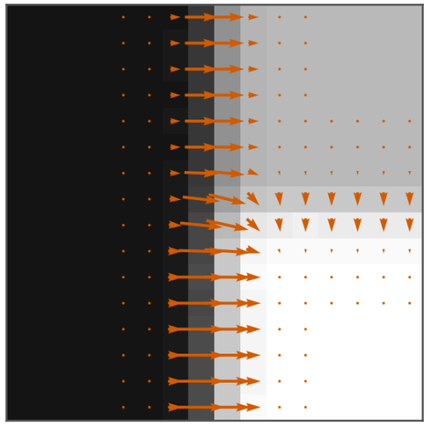
View B is rotated 90° and enlarged $2\times$. After alignment, both patches have nearly the same appearance and gradient directions.

Compute the Image Patch's Gradient

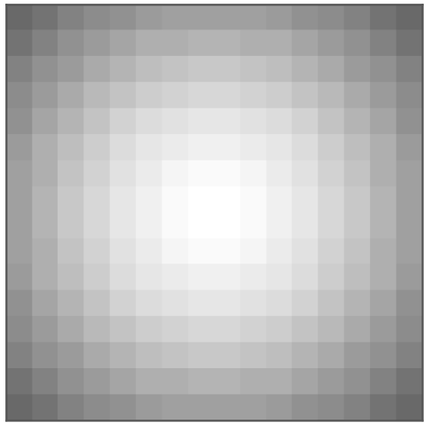
- Compute patch gradient, and use Gaussian weights to reduce the influence of pixels far from the keypoint.
- Gradient **angles stay the same; only magnitudes change** with Gaussian weighting.

```
sf_m, sf_a = sf_gradients(sf_canon)
sf_g = cv2.getGaussianKernel(16, 8)
sf_weight = sf_g @ sf_g.T
sf_weight /= sf_weight.max()
sf_votes = sf_m * sf_weight
```

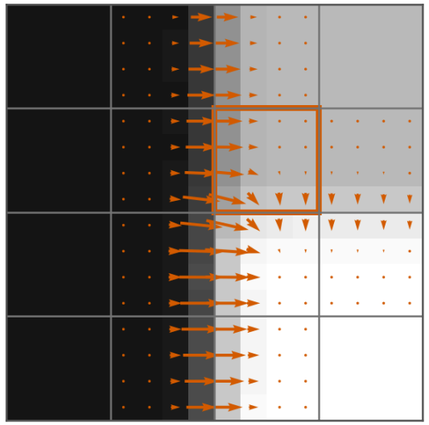
Before weighting



Gaussian weights



After weighting



- **Divide the aligned patch into 4×4 cells.**
- Let's inspect the highlighted cell in **row 2, column 3** next.

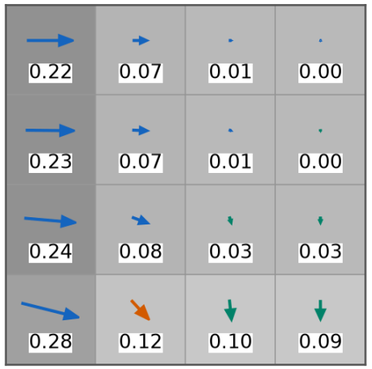
One Cell: Gradient Arrows Become Eight Numbers

- Divide the 360° orientation range into eight bins centered at $0^\circ, 45^\circ, \ldots, 315^\circ$.
- Each pixel's gradient **angle selects the nearest bin**, and its weighted gradient **magnitude adds to that bin's total**.
- **Add magnitudes, not angles!**

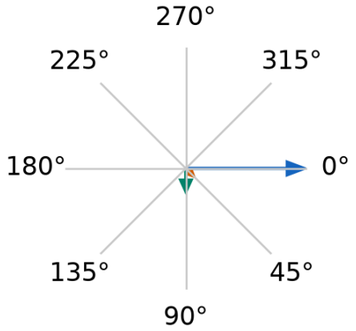
```python
def sf_hist8(angles, votes):
    bins = ((angles + 22.5) // 45).astype(int) % 8
    return np.bincount(bins.ravel(), weights=votes.ravel(), minlength=8)

sf_cell_angles = sf_a[4:8, 8:12]
sf_cell_votes = sf_votes[4:8, 8:12]
sf_cell_hist = sf_hist8(sf_cell_angles, sf_cell_votes)
```

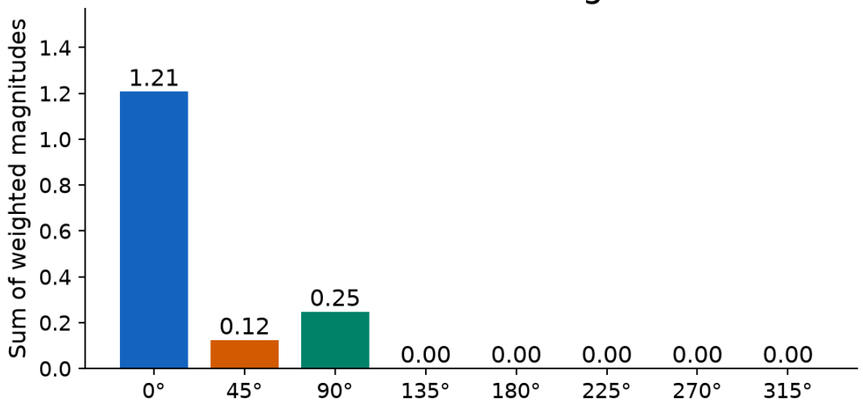
16 pixel votes



Sum votes by direction



The same sums as eight bars



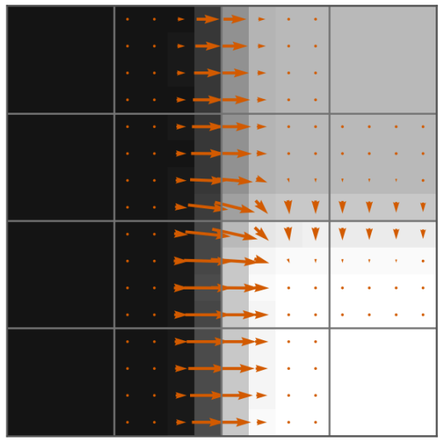
Cell (2, 3): [1.21, 0.12, 0.25, 0.00, 0.00, 0.00, 0.00, 0.00]

Sixteen Cells: 128 Numbers

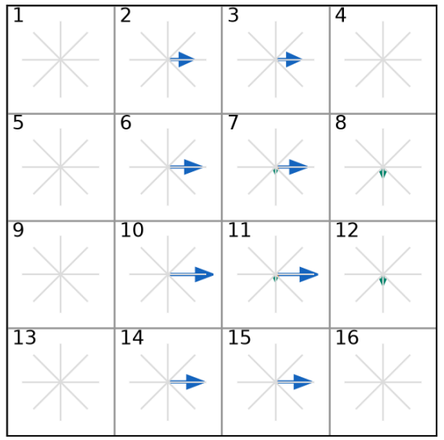
- Repeat the same as before in every cell.

```
sf_H = np.zeros((4, 4, 8))
for row in range(4):
    for col in range(4):
        region = np.s_[4*row:4*row+4, 4*col:4*col+4]
        sf_H[row, col] = sf_hist8(sf_a[region], sf_votes[region])
sf_raw = sf_H.ravel()                          # 16 cells × 8 bins = 128 values
```

Pixel gradients in 16 cells



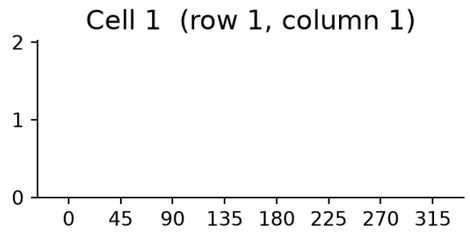
16 orientation histograms



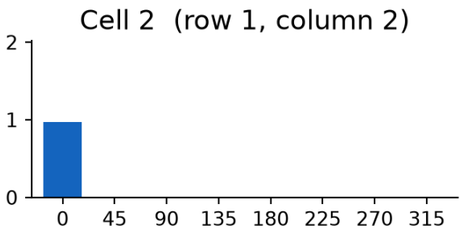
Each group of eight arrows represents one local histogram. Longer arrows mean larger bin sums. Flat cells contribute zeros. Read cells left to right, then top to bottom.

The Descriptor Values: Cells 1–8

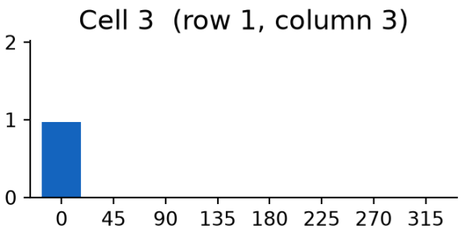
Each bar becomes one number. Read the bins in the order $0^\circ, 45^\circ, \ldots, 315^\circ$.



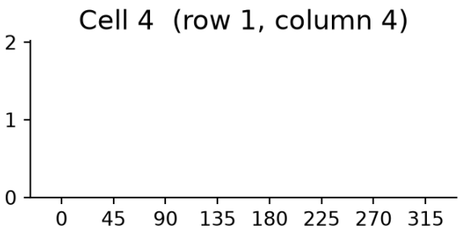
[0, 0, 0, 0, 0, 0, 0, 0]



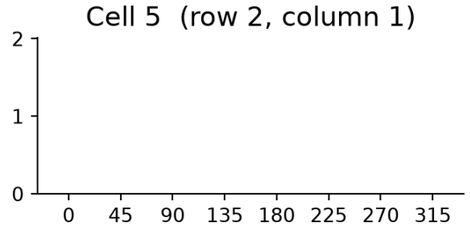
[0.97, 0, 0, 0, 0, 0, 0, 0]



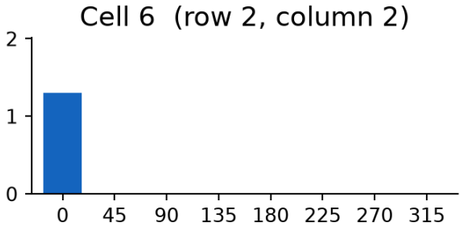
[0.97, 0, 0, 0, 0, 0, 0, 0]



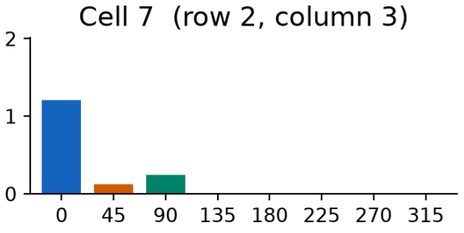
[0, 0, 0, 0, 0, 0, 0, 0]



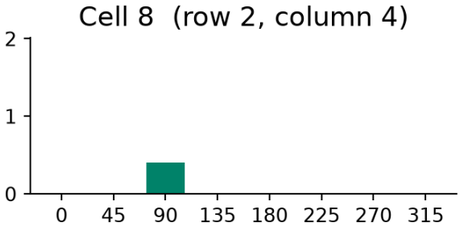
[0, 0, 0, 0, 0, 0, 0, 0]



[1.3, 0, 0, 0, 0, 0, 0, 0]



[1.21, 0.12, 0.25, 0, 0, 0, 0, 0]

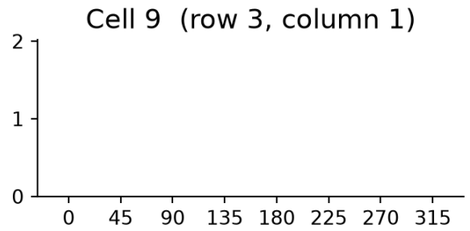


[0, 0, 0.4, 0, 0, 0, 0, 0]

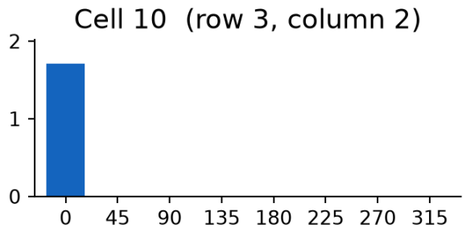
The vectors beneath these eight histograms supply descriptor components **1–64**.

The Descriptor Values: Cells 9–16

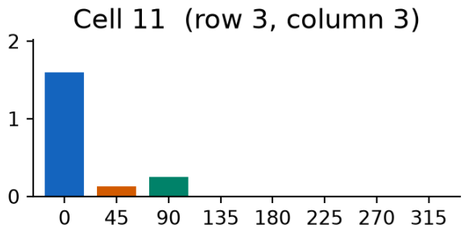
Continue in the same order, appending eight numbers from each remaining cell.



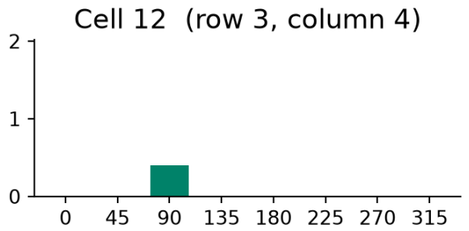
[0, 0, 0, 0, 0, 0, 0, 0]



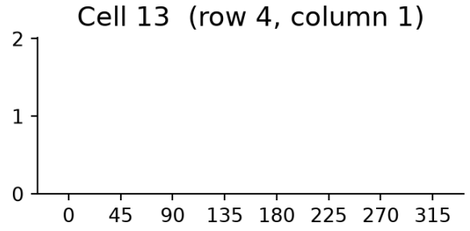
[1.71, 0, 0, 0, 0, 0, 0, 0]



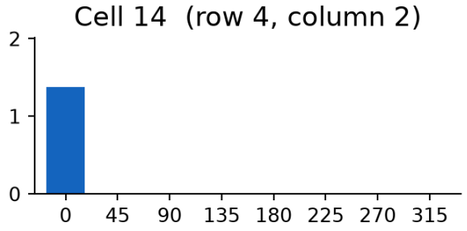
[1.6, 0.13, 0.25, 0, 0, 0, 0, 0]



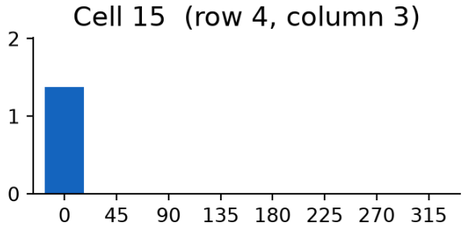
[0, 0, 0.4, 0, 0, 0, 0, 0]



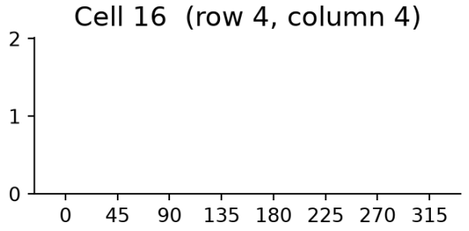
[0, 0, 0, 0, 0, 0, 0, 0]



[1.38, 0, 0, 0, 0, 0, 0, 0]



[1.38, 0, 0, 0, 0, 0, 0, 0]



[0, 0, 0, 0, 0, 0, 0, 0]

$$\mathbf{d}_{\text{raw}} = [\ \underbrace{\mathbf{h}_1}_{8\text{ values}}, \underbrace{\mathbf{h}_2}_{8\text{ values}}, \ldots, \underbrace{\mathbf{h}_{16}}_{8\text{ values}}\ ] \in \mathbb{R}^{128}.$$

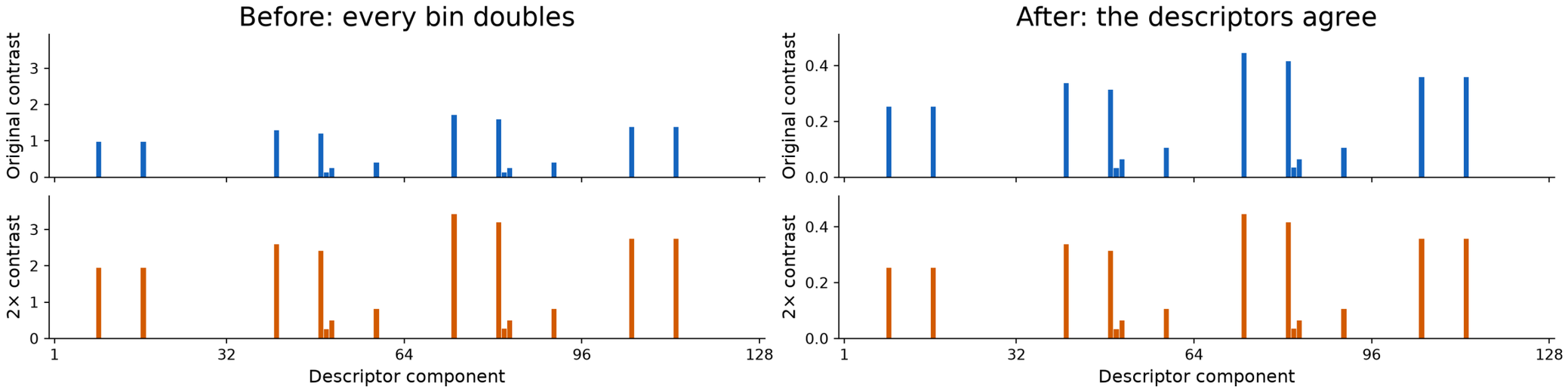
These eight vectors supply components **65–128**.

Normalize the Descriptor

- For an intensity change $I' = aI + b$ with $a > 0$, the gradient magnitudes and histogram sums scale by a.
- The derivative removes b. Dividing by the vector's length removes the common positive factor a.

```python
def sf_unit(vector):
    return vector / max(np.linalg.norm(vector), 1e-12)

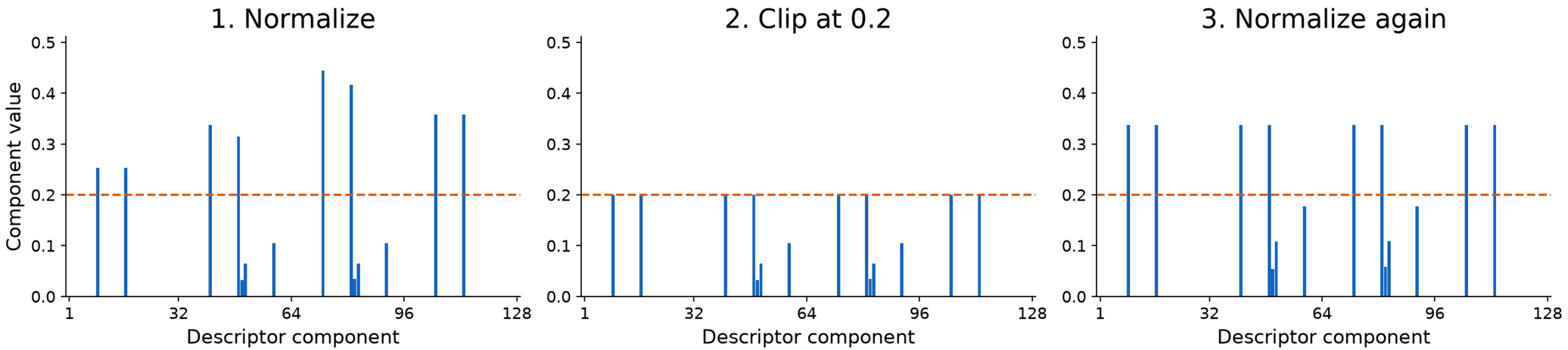
sf_normalized = sf_unit(sf_raw)
sf_gain_normalized = sf_unit(2*sf_raw)
```



Limit the Influence of Strong Gradients

Clip unusually large components, then normalize again (so that a few very strong bins do not dominate the descriptor).

```
sf_clipped = np.minimum(sf_normalized, 0.2)
sf_descriptor = sf_unit(sf_clipped)
```



- The final vector has **unit length**.
- Its components can exceed 0.2 again after the second normalization.

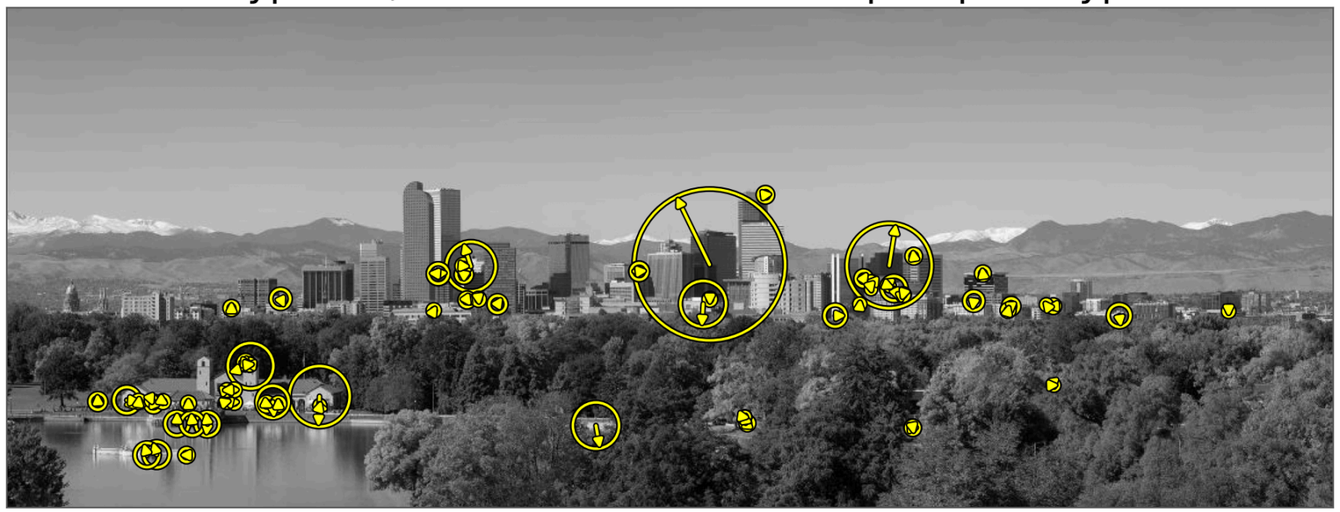
From the Teaching Example to Full SIFT in OpenCV

Full SIFT refines extrema and orientation peaks, shares votes across neighboring cells and bins, and can retain multiple orientations. Our teaching example used hard bins and one orientation.

```
sf_sift = cv2.SIFT_create(nfeatures=80)
sf_keypoints, sf_descriptors = sf_sift.detectAndCompute(sf_image, None)

# Each keypoint provides: .pt, .size, and .angle (in degrees).
```

80 keypoints; one 128-number descriptor per keypoint



Circles indicate keypoint scale; arrows show the assigned orientation. Symbols are enlarged $4\times$ for visibility. OpenCV SIFT documentation

Local Feature Matching: Main Components

We have locations, scales, orientations, and descriptors. Now, find corresponding features.

Image A

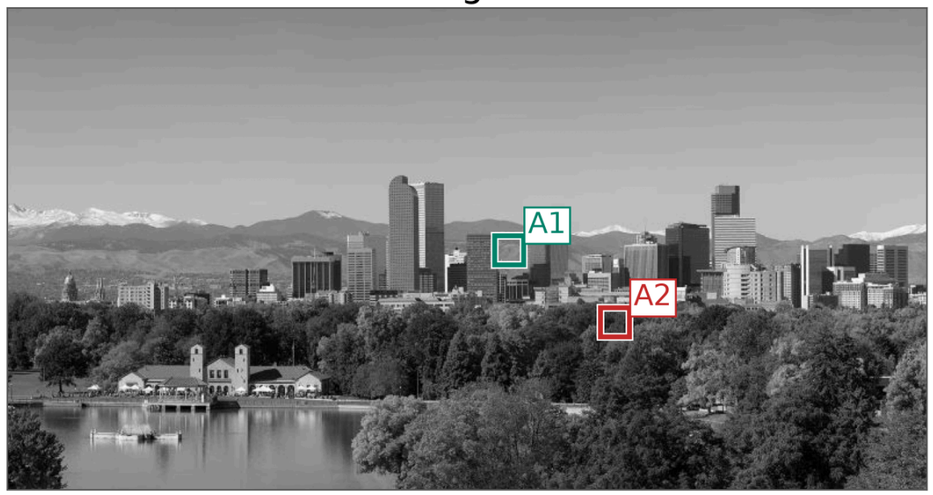
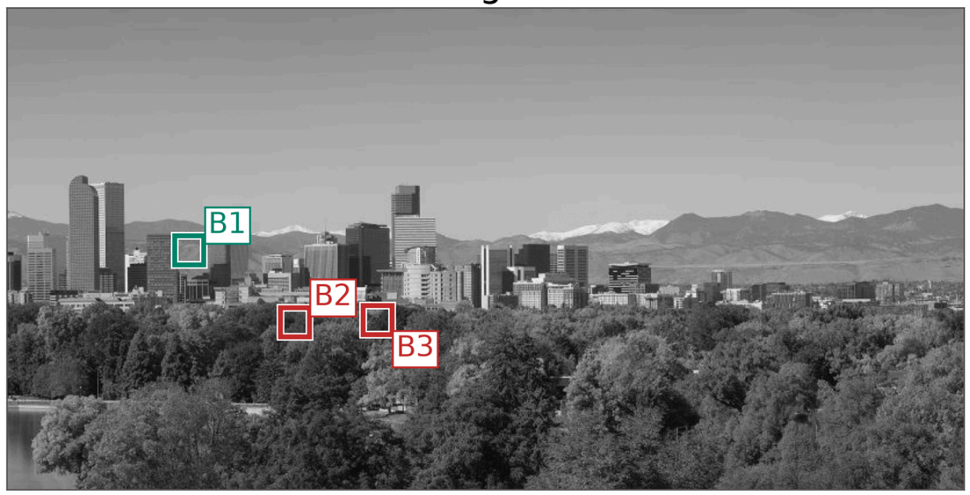


Image B



1. Detection

Find distinctive locations and their scales.

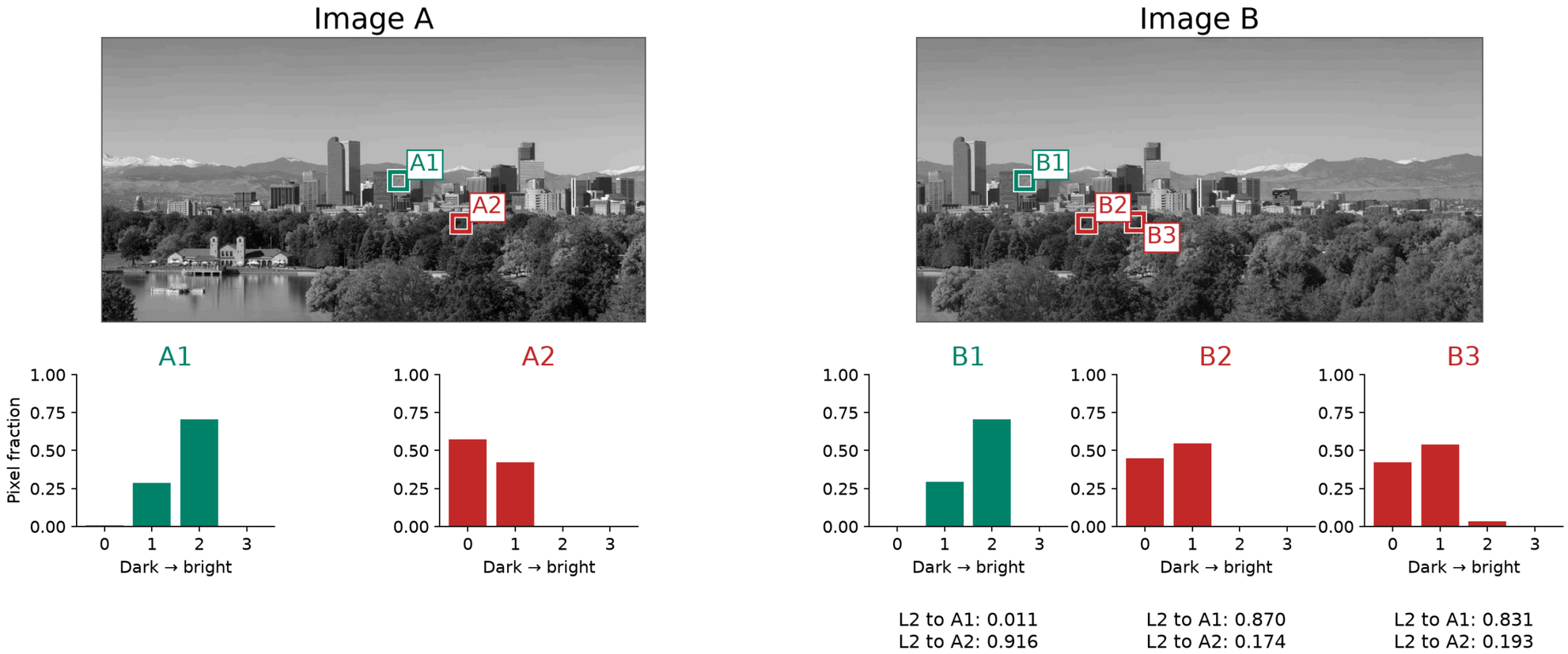
2. Description

Encode each neighborhood as a vector.

3. Matching

Compare descriptors to find correspondences.

Which Features Would You Match?



- Which candidate would you choose for **A1** and **A2**?
- Would you trust both choices equally?

Distance Between Descriptors

Euclidean distance: separation

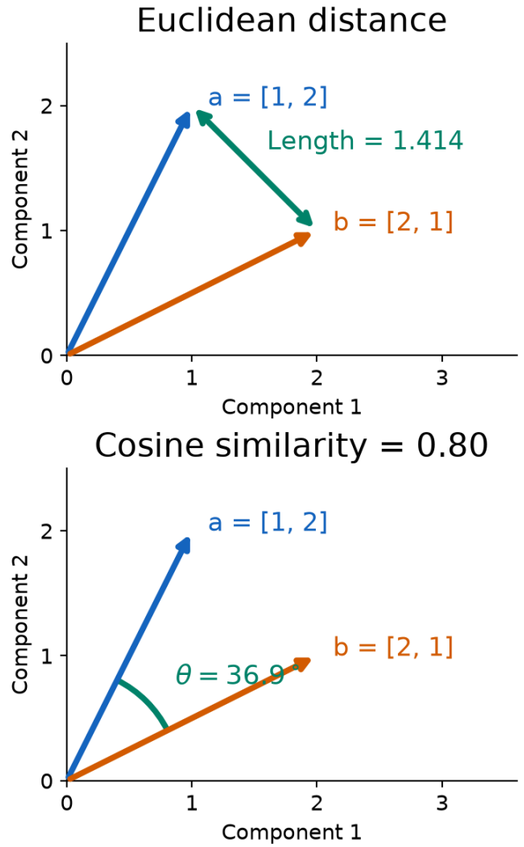
$$d_2(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_i (a_i - b_i)^2}.$$

The arrow connecting the vector tips has length d_2.
Smaller is closer.

Cosine similarity: direction

$$s_{\cos} = \frac{\mathbf{a}^\top \mathbf{b}}{\|\mathbf{a}\|_2 \|\mathbf{b}\|_2} = \cos\theta.$$

Larger is more similar. Cosine distance is $1 - s_{\cos}$.



For unit-length vectors, $d_2^2 = 2(1 - s_{\cos})$: L2 and cosine give the **same nearest-neighbor ordering**.

Distances and Similarity for Descriptors

Let $\mathbf{p} = [0.2, 0.8]$ and $\mathbf{q} = [0.4, 0.6]$ be histograms that sum to one.

Measure	Definition	Example
Euclidean / L2	$\sqrt{\sum_i (p_i - q_i)^2}$	0.2828
Cosine similarity	$s_{\cos} = \frac{\mathbf{p}^\top \mathbf{q}}{\|\mathbf{p}\|_2 \|\mathbf{q}\|_2}$	0.9417
Cosine distance	$1 - s_{\cos}$	0.0583
Manhattan / L1	$\sum_i \|p_i - q_i\|$	0.4000
Chi-square	$\frac{1}{2} \sum_i \frac{(p_i - q_i)^2}{p_i + q_i}$	0.0476
Hellinger	$\frac{1}{\sqrt{2}} \|\sqrt{\mathbf{p}} - \sqrt{\mathbf{q}}\|_2$	0.1560

```python
p, q = np.array([0.2, 0.8]), np.array([0.4, 0.6])
euclidean = np.linalg.norm(p-q)
cosine_similarity = p @ q / (np.linalg.norm(p)*np.linalg.norm(q))
cosine_distance = 1-cosine_similarity
manhattan = np.abs(p-q).sum()
chi_square = 0.5*np.sum((p-q)**2 / np.maximum(p+q, 1e-12))
hellinger = np.linalg.norm(np.sqrt(p)-np.sqrt(q)) / np.sqrt(2)
```

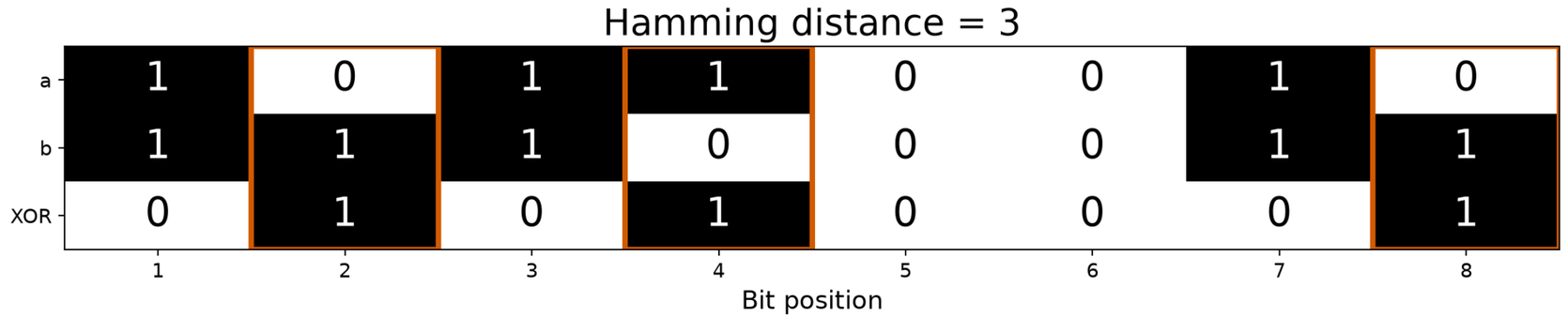
Hellinger assumes nonnegative histograms that sum to one; a zero/zero chi-square term contributes zero. **Standard SIFT matching uses L2.**

Binary Descriptors: Hamming Distance

Some descriptors are binary (such as ORB). Hamming distance counts the positions where the bits disagree:

$$d_H(\mathbf{a}, \mathbf{b}) = \sum_i \mathbf{1}[a_i \neq b_i]$$

```
a = np.array([1,0,1,1,0,0,1,0], dtype=np.uint8)
b = np.array([1,1,1,0,0,0,1,1], dtype=np.uint8)
different = a ^ b                          # bitwise XOR
hamming = np.count_nonzero(different)      # 3 differing bits
```



OpenCV uses **NORM_HAMMING** for default ORB. We count differing bits, not differing packed bytes.

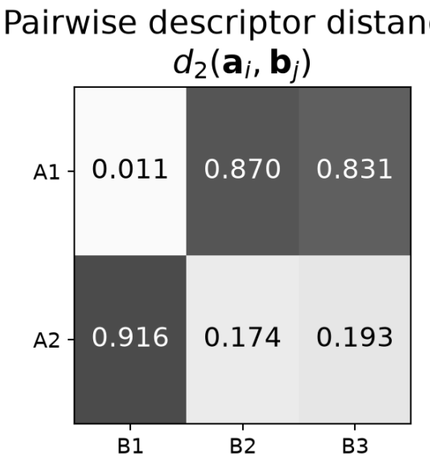
Efficient Comparison: A Matrix of Distances

With N descriptors in A and M in B, exhaustive matching compares NM pairs.

$$d_2(\mathbf{a}_i, \mathbf{b}_j) = \sqrt{\|\mathbf{a}_i\|_2^2 + \|\mathbf{b}_j\|_2^2 - 2\mathbf{a}_i^\top \mathbf{b}_j}.$$

```
norms_a = np.sum(descriptors_a**2, axis=1)  # one descriptor per row
norms_b = np.sum(descriptors_b**2, axis=1)
squared_distances = norms_a[:, None] + norms_b[None, :] - 2*descriptors_a @ descriptors_b.T
distances = np.sqrt(np.maximum(squared_distances, 0))  # guard against rounding
```

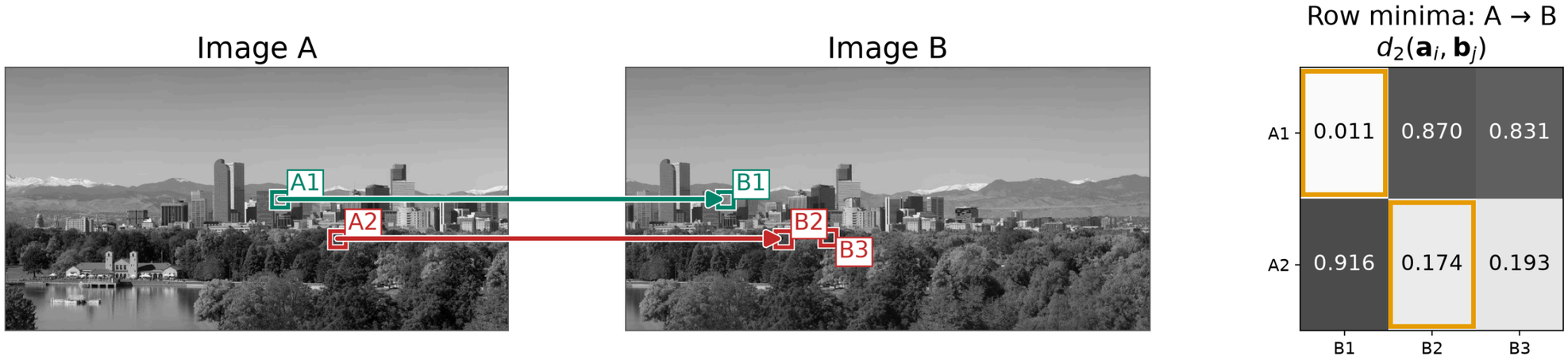
- For **L2-unit** descriptors, both squared norms are one.
- Matrix operations avoid Python loops, but still require $O(NMK)$ work for K components.
- Use batches or approximate search for large collections.



Nearest Neighbor: The Closest Descriptor

The **nearest neighbor** minimizes descriptor distance: $j^* = \arg\min_j d_2(\mathbf{a}_i, \mathbf{b}_j)$.

```
neighbor_order = np.argsort(distances, axis=1)  # sort each row: A → B
nearest_b = neighbor_order[:, 0]
sorted_distances = np.take_along_axis(distances, neighbor_order, axis=1)
nearest_distances, second_distances = sorted_distances[:, 0], sorted_distances[:, 1]
```

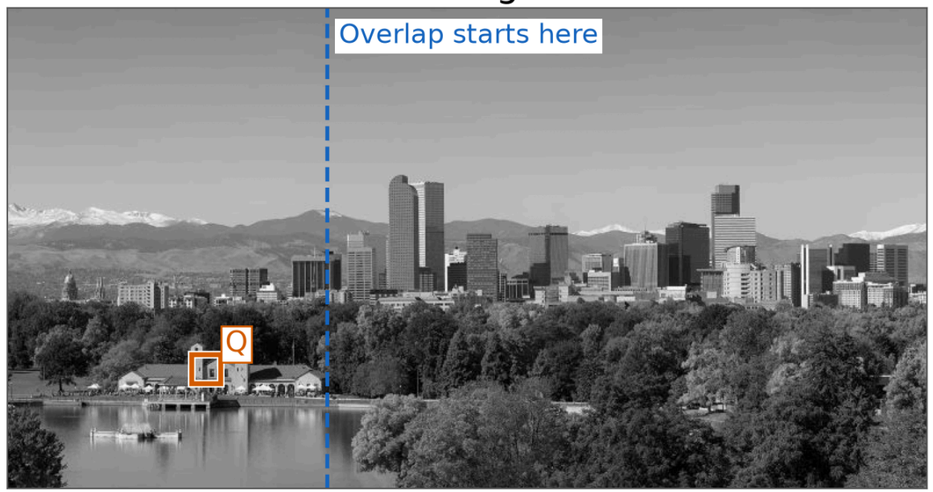


- **Second and third nearest neighbors** have the second- and third-smallest distances.
- "Nearest" refers to **descriptor space**, not pixel coordinates.

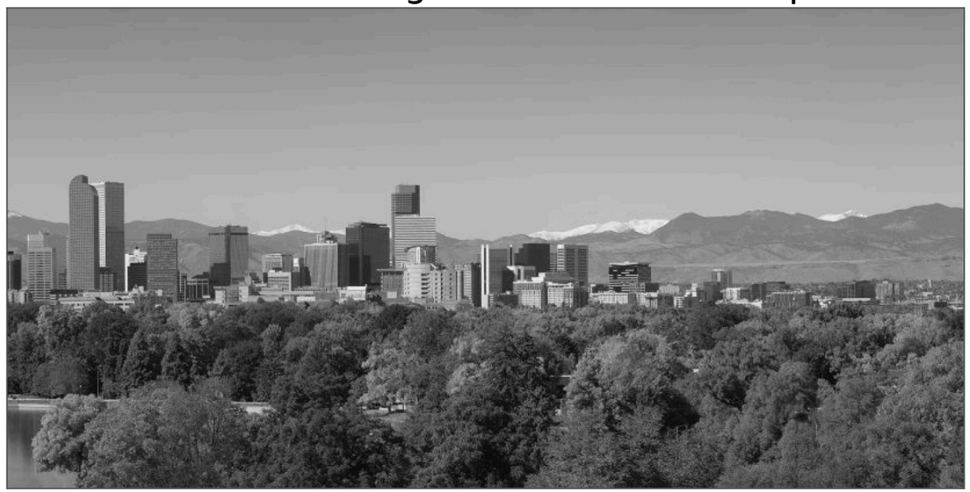
Challenge 1: A Feature May Have No Counterpart

A **nearest-neighbor search always returns a candidate** if the target set is nonempty, even when the scene point is absent.

A: this building is visible



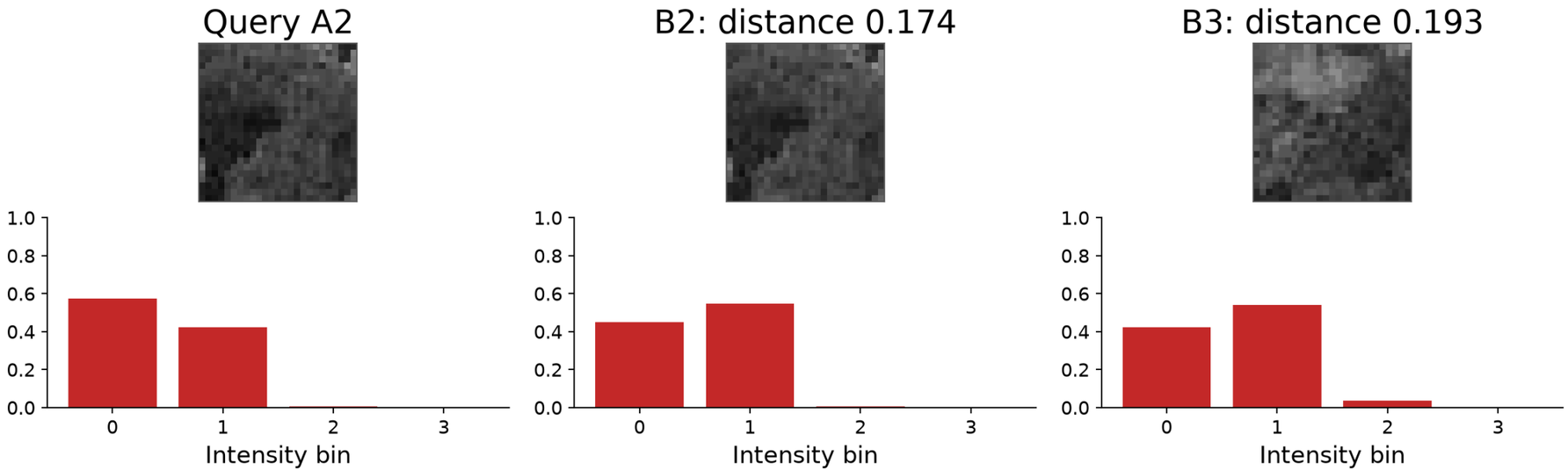
B: the building lies outside the crop



- Features can disappear through lack of overlap, occlusion, or failed detection.
- We need a way to **reject** a proposed match.

Challenge 2: Several Candidates Can Look Similar

An absolute distance threshold accepts the nearest candidate only when $d_1 < T$. A small distance alone does not establish uniqueness.



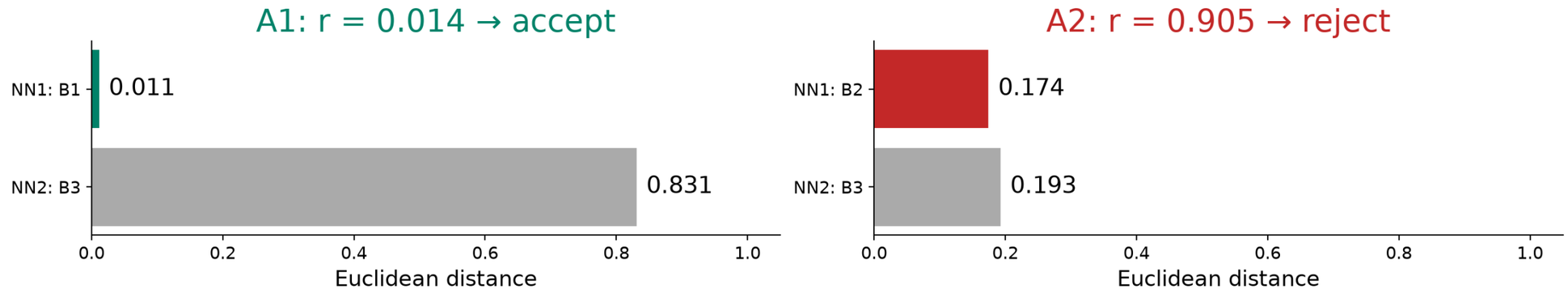
- The cutoff depends on the descriptor and its normalization.
- With $T = 0.20$, both candidates are close: **why trust B2 when B3 looks almost as good?**

Lowe’s Ratio Test: Is the Best Match Distinctive?

Compare distances to the nearest and second-nearest candidates:

$$r = \frac{d_1}{d_2}, \qquad \text{accept the nearest match if } d_1 < \tau d_2.$$

```
ratio_threshold = 0.8
distance_ratios = nearest_distances / np.maximum(second_distances, 1e-12)
ratio_pass = nearest_distances < ratio_threshold*second_distances
```



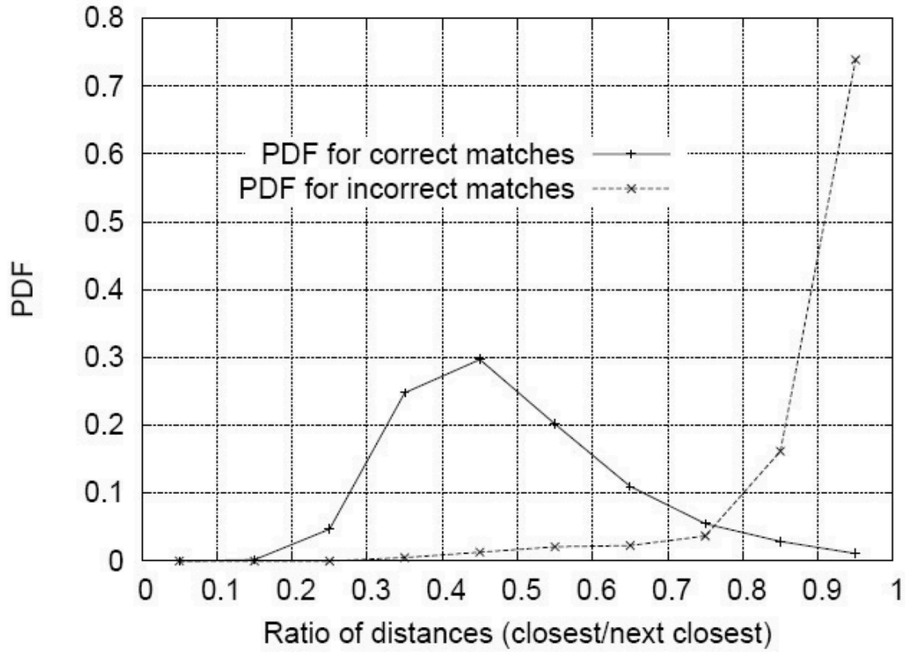
- A **small ratio** means the winner stands out.
- A **ratio near one** means **ambiguity**. If both distances are zero, the strict test rejects the tie.

How Did Lowe Choose the Threshold?

Lowe measured how well the ratio separates correct and incorrect matches.

Experiment

- Match transformed images against a database of **40,000 keypoints**.
- Use correspondence information to label nearest matches correct or incorrect.
- Compare their ratio distributions and choose a cutoff balancing errors.



Reported at a cutoff of 0.8

- Reject about **90% of false matches**.
- Lose **less than 5% of correct matches**.

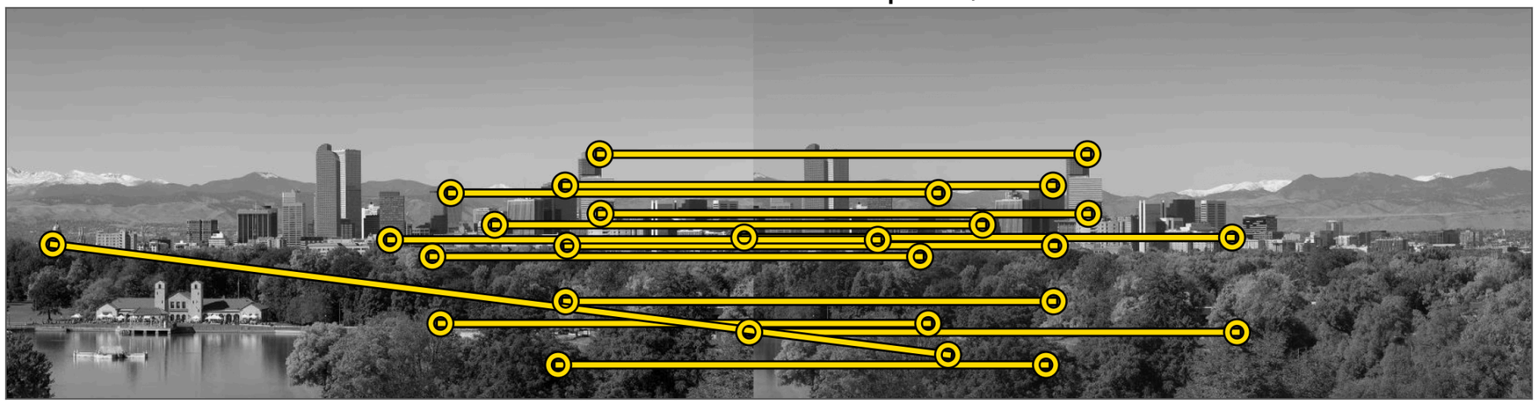
Lowe (2004), Figure 11 and §7.1: random scale/rotation changes, 30° depth rotation, and 2% image noise. Original paper

SIFT Matching in OpenCV

Find the two closest SIFT descriptors in Image B for each descriptor in Image A, then apply the ratio test.

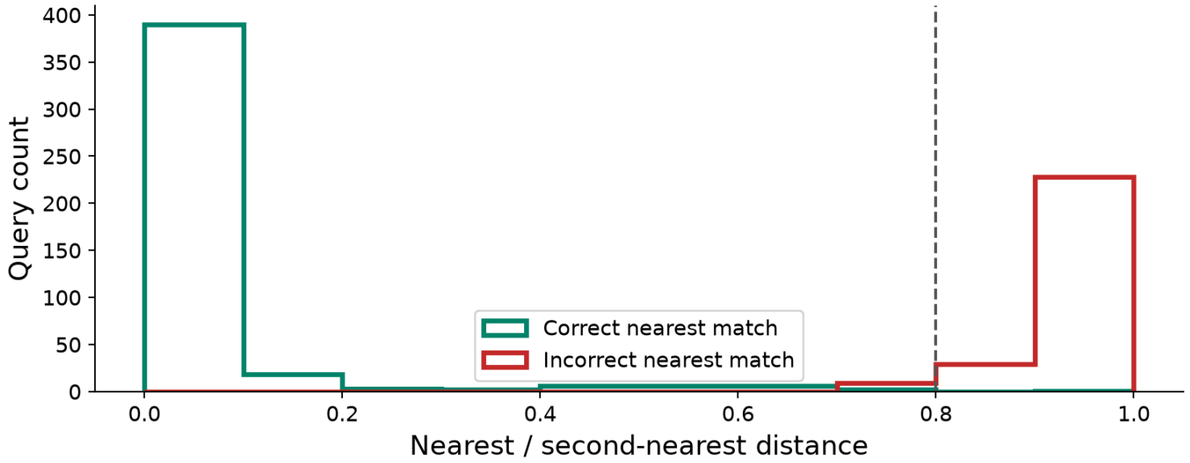
```
sift = cv2.SIFT_create(nfeatures=700)
keypoints_a, sift_descriptors_a = sift.detectAndCompute(image_a, None)
keypoints_b, sift_descriptors_b = sift.detectAndCompute(image_b, None)
neighbors_ab = cv2.BFMatcher(cv2.NORM_L2).knnMatch(sift_descriptors_a, sift_descriptors_b, k=2
neighbors_ab = [pair for pair in neighbors_ab if len(pair) == 2]
ratio_matches_ab = [m for m, n in neighbors_ab if m.distance < 0.8*n.distance]
```

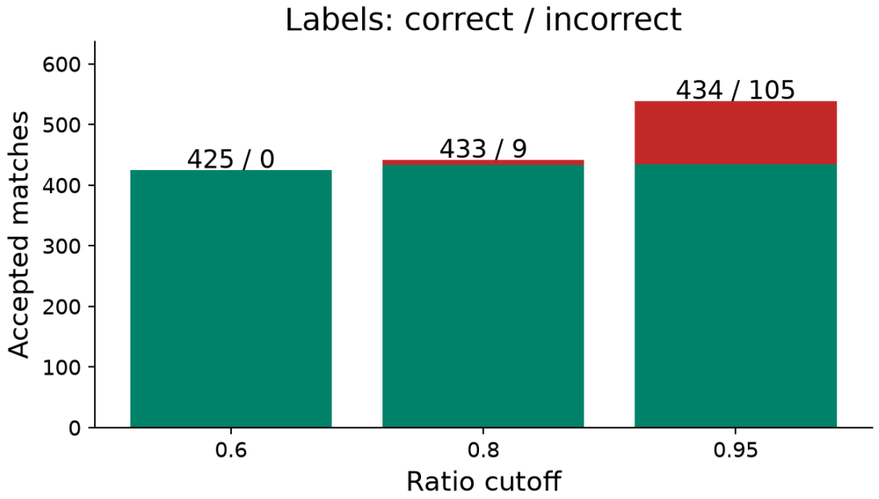
Forward ratio test: 442 accepted; 14 shown



- The ratio test produces **tentative correspondences**.
- Further verification is still needed to reject wrong matches.

Choosing a Threshold

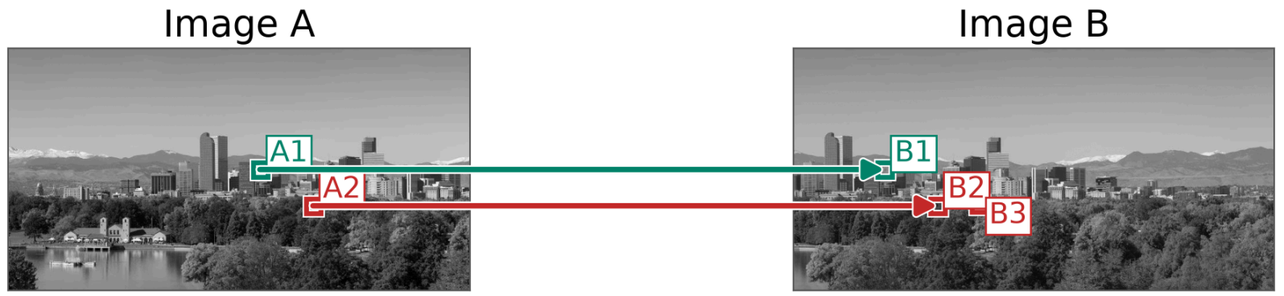


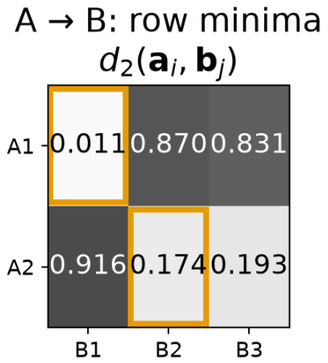


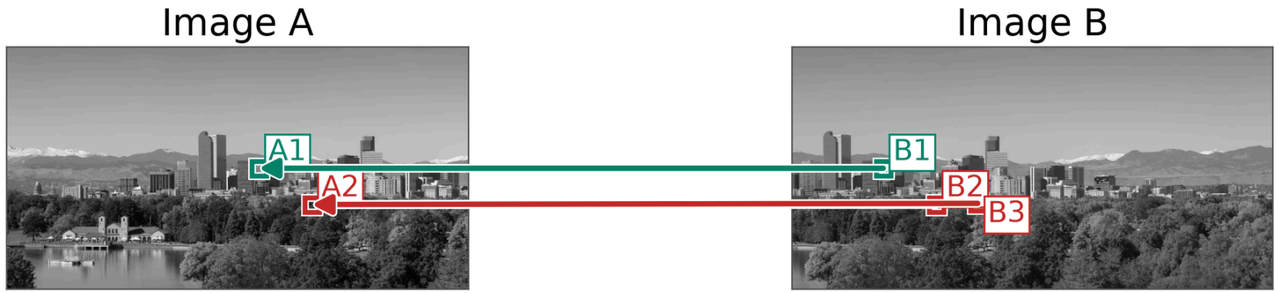
- Here, a match is correct if the known translation aligns keypoints within 3 pixels.
- Lowe's original 2004 SIFT paper used 0.8, but **there is no guarantee for correct matches.**
- In practice, people use values from about 0.6 to 0.8:
  - 0.7–0.75 is the most common default, including many OpenCV examples.
  - 0.6–0.65 is stricter and gives fewer matches with higher precision.
  - 0.8 keeps more matches.
- Downstream robust estimators such as RANSAC or graph-based outlier rejection can be used subsequently to reject outlier matches.

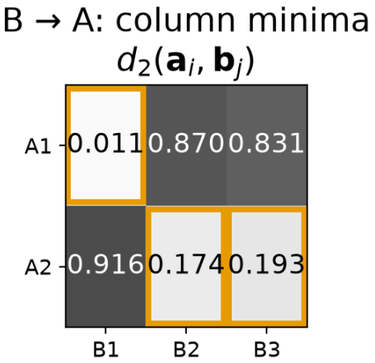
Bidirectional Matching: Do the Choices Agree?

```python
nearest_b = np.argmin(distances, axis=1)  # [B1, B2] for A1, A2
nearest_a = np.argmin(distances, axis=0)  # [A1, A2, A2] for B1, B2, B3
```









Keep a pair only if both features choose each other: A1 ↔ B1 and A2 ↔ B2. B3 chooses A2, but A2 chooses B2.

Combine Mutual Matching with Lowe's Ratio Test

Keep **mutual nearest neighbors** that pass the ratio test in **both directions**.

```python
def ratio_filter(neighbors, threshold=0.8):
    accepted = {}
    for pair in neighbors:
        if len(pair) == 2:
            first, second = pair
            if first.distance < threshold*second.distance:
                accepted[first.queryIdx] = first.trainIdx
    return accepted
matcher = cv2.BFMatcher(cv2.NORM_L2)
forward = ratio_filter(matcher.knnMatch(sift_descriptors_a, sift_descriptors_b, k=2))
reverse = ratio_filter(matcher.knnMatch(sift_descriptors_b, sift_descriptors_a, k=2))
mutual_pairs = [(i, j) for i, j in forward.items() if reverse.get(j) == i]
```

- A1 ↔ B1: mutual and distinctive → **keep**.
- A2 ↔ B2: mutual, but forward ratio 0.905 → **reject**.

These are still **tentative correspondences**.

Toy example: only A1 ↔ B1 survives

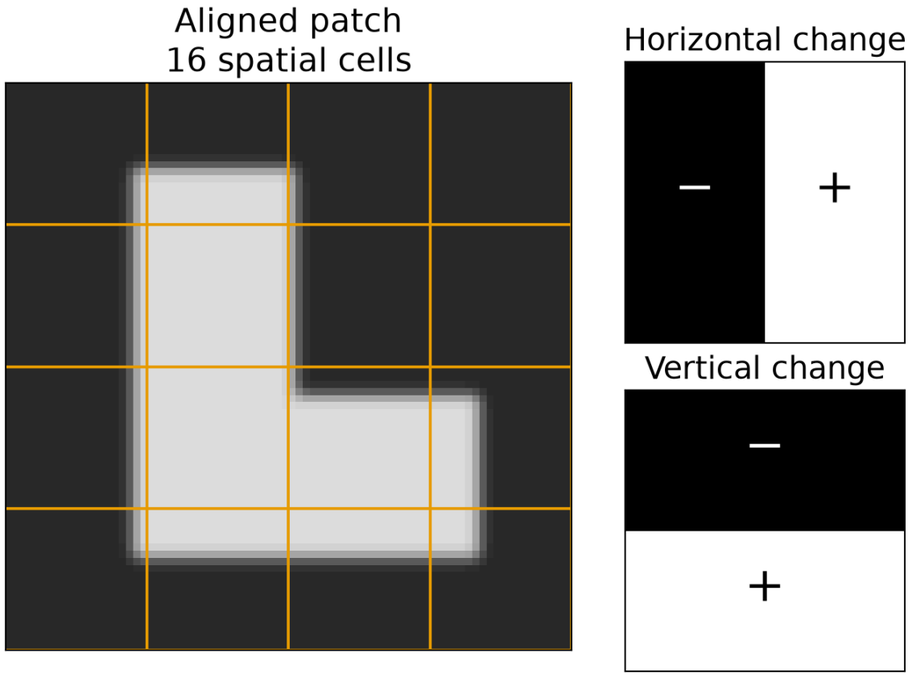
Pair	Mutual?	Both ratios?	Decision
A1 ↔ B1	Yes	Yes	Keep
A2 ↔ B2	Yes	No	Reject

SURF: Speeded-Up Robust Features

SURF uses rectangular filters to reduce the work of detection and description.

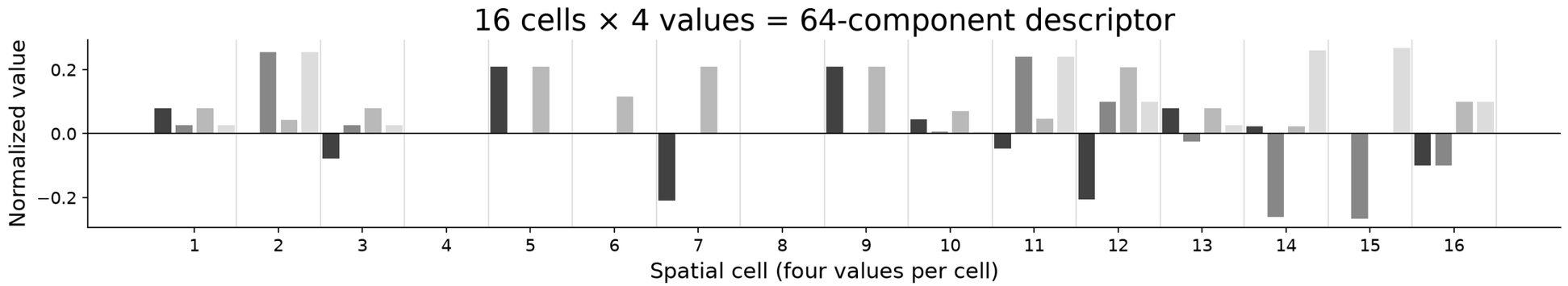
- **Detection:** find blob-like features across scales using rectangular approximations to second-derivative filters.
- **Description:** align a patch to its scale and orientation, then summarize horizontal and vertical changes in a 4×4 grid.
- **Why useful:** compact and efficient features for image matching and object recognition.

Integral images (to be defined later) make rectangular sums inexpensive, even for large filters.



Bay, Tuytelaars, and Van Gool (2006), SURF.

SURF: Four Numbers per Cell



- Each cell stores $[\sum d_x,\ \sum d_y,\ \sum |d_x|,\ \sum |d_y|]$. Signed sums capture direction; absolute sums retain changes that would cancel.
- Full SURF also uses scale/orientation alignment and Gaussian weighting.
- Its standard descriptor has **64 components**, compared with SIFT's 128, and uses **L2 matching**.

```
surf = cv2.xfeatures2d.SURF_create(hessianThreshold=400, extended=False)
surf_keypoints, surf_descriptors = surf.detectAndCompute(skyline_gray, None)
surf_matcher = cv2.BFMatcher(cv2.NORM_L2)
```

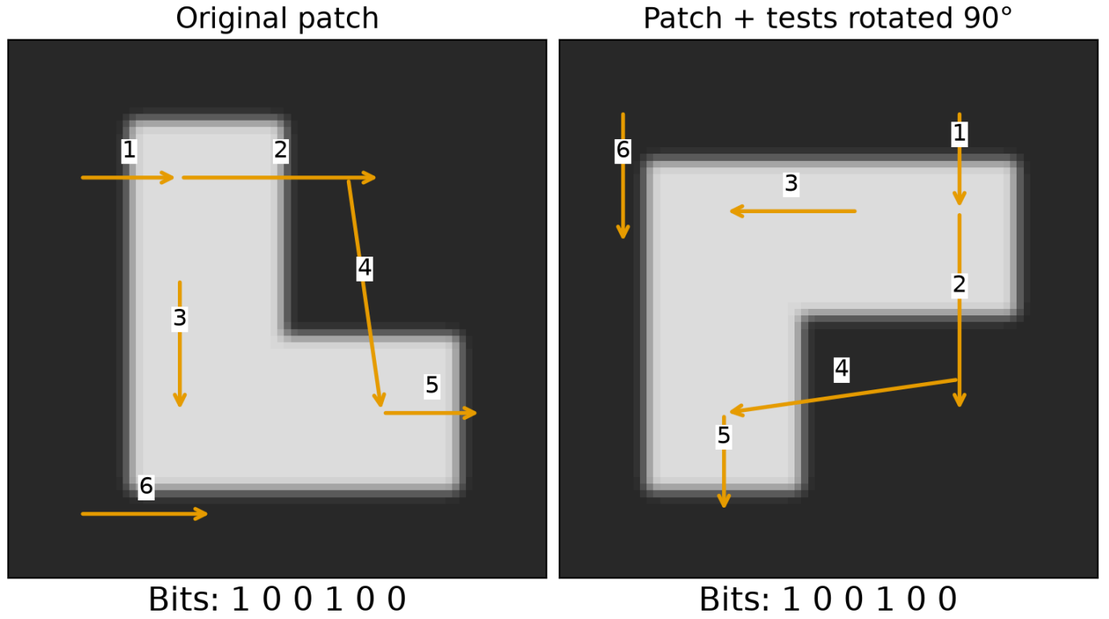
The SURF call requires OpenCV contrib built with `OPENCV_ENABLE_NONFREE=ON`.

ORB: Oriented FAST and Rotated BRIEF

- Detect **FAST corners** in an image pyramid and rank them using Harris scores.
- Estimate orientation from the patch's intensity centroid.
- Rotate a pattern of pixel-pair tests to this orientation. Each brightness comparison gives **one bit**.

$$b_k = \begin{cases} 1, & I(\mathbf{p}_k) < I(\mathbf{q}_k), \\ 0, & \text{otherwise.} \end{cases}$$

- Standard ORB uses **256 bits** and **Hamming distance**. Compact descriptors suit fast matching and visual SLAM.



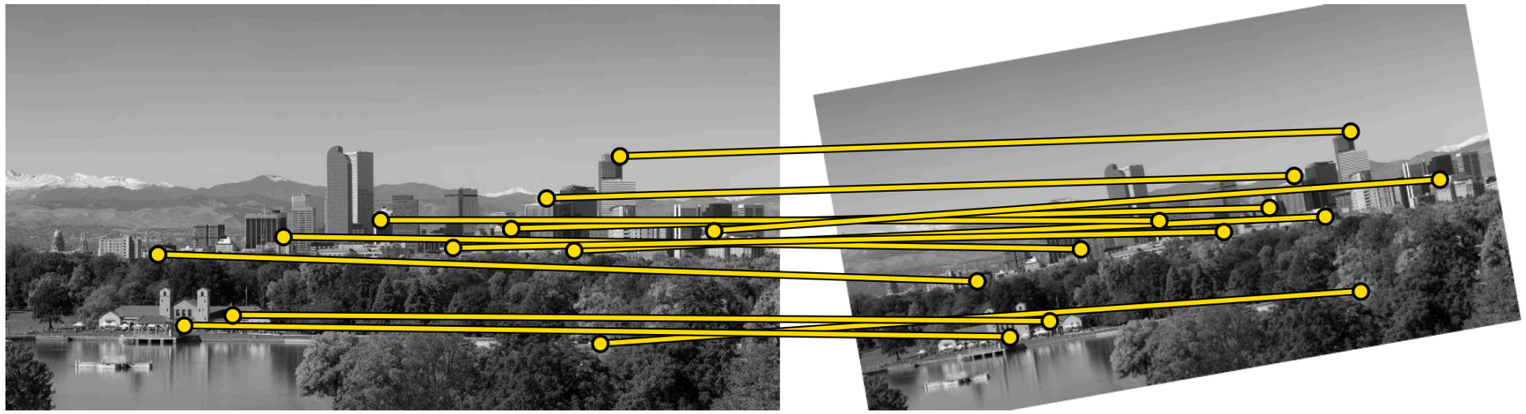
Rotating the sampling pattern preserves these bits in the example.

Rublee et al. (2011). OpenCV ORB explanation.

ORB in OpenCV: Binary Descriptors and Matches

```
1 orb = cv2.ORB_create(nfeatures=700)
2 orb_keypoints_a, orb_descriptors_a = orb.detectAndCompute(orb_image_a, None)
3 orb_keypoints_b, orb_descriptors_b = orb.detectAndCompute(orb_image_b, None)
4 orb_neighbors = cv2.BFMatcher(cv2.NORM_HAMMING).knnMatch(orb_descriptors_a, orb_descriptors_b,
5 orb_neighbors = [pair for pair in orb_neighbors if len(pair) == 2]
6 orb_matches = [first for first, second in orb_neighbors if first.distance < .75*second.distanc
```

425 tentative matches; 12 shown

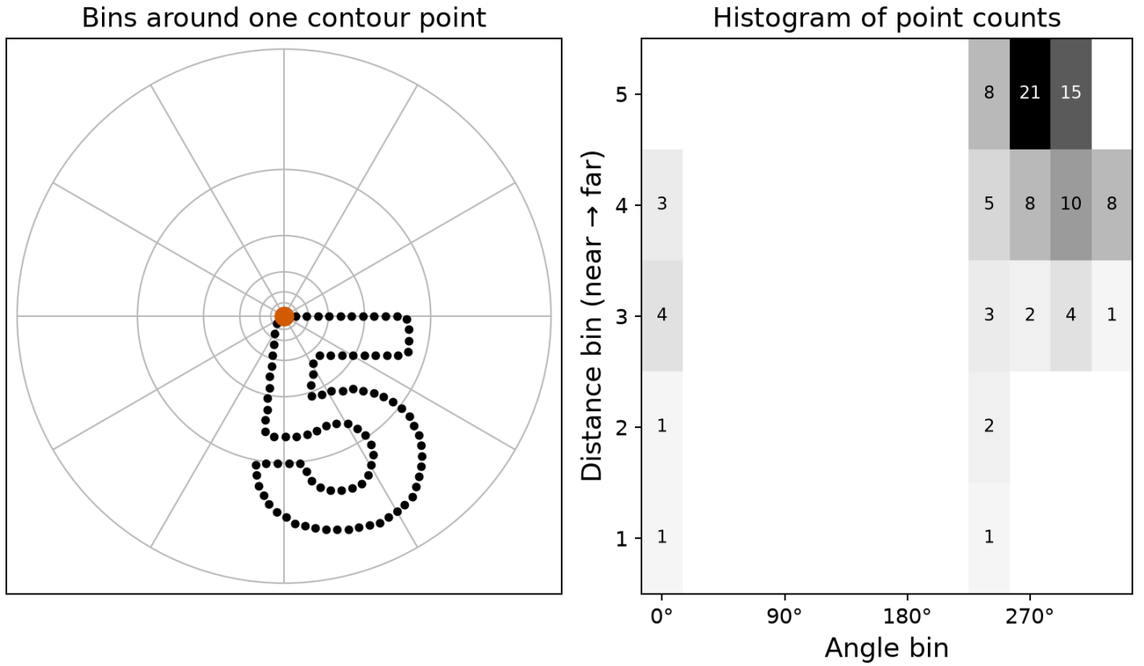


- Each descriptor occupies **32 bytes = 256 bits**.
- We still use the ratio test, but with **Hamming distances**.

Shape Context: The Arrangement of Contour Points

- Count points in bins of **direction and distance**.
- Use logarithmic radial bins for finer detail nearby and coarser detail farther away.
- Repeat at every sampled point to describe the whole shape.

Useful for silhouettes, characters, and shapes whose texture is uninformative.



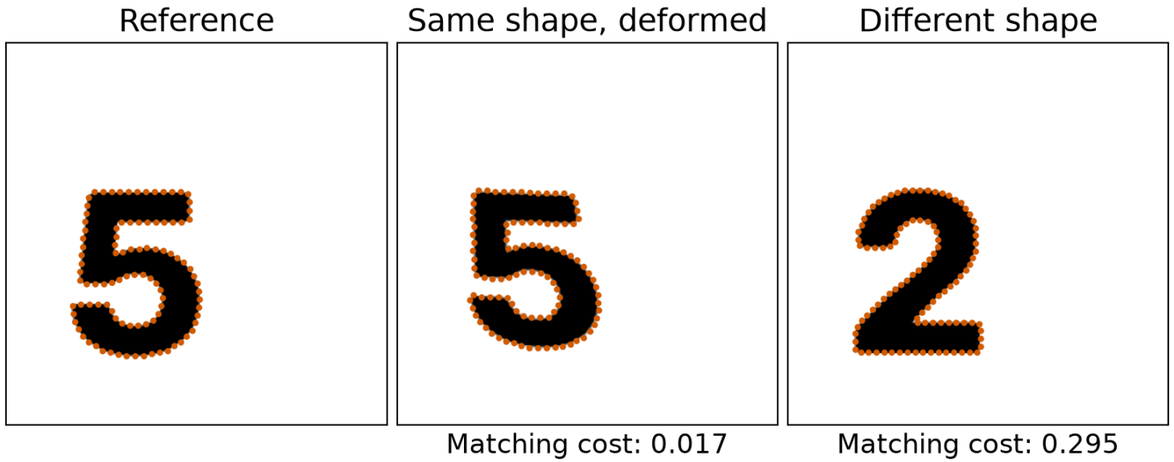
Belongie, Malik, and Puzicha (2002), Shape Matching and Object Recognition Using Shape Contexts.

Shape Context: Matching Shapes in OpenCV

Compare point histograms, find correspondences, then refine a deformable alignment.

```
shape_matcher = cv2.createShapeContextDistanceExtractor()
same_shape_cost = shape_matcher.computeDistance(contour_a, contour_b)
different_shape_cost = shape_matcher.computeDistance(contour_a, contour_c)
```

- **Chi-square costs** compare the histograms at candidate contour points.
- A smaller overall cost means better descriptor agreement and alignment.
- Useful for moderate deformation, but missing boundaries and clutter can disrupt matching.



Requires OpenCV's **contrib shape module**. OpenCV documentation

Key Takeaways

- Good features are repeatable across images and distinctive enough to recognize again. Detection finds their locations; descriptors represent their appearance.
- Harris detects corners where intensity changes strongly in two independent directions. The eigenvalues of the second moment matrix distinguish flat regions, edges, and corners.
- A fixed analysis scale does not handle all feature sizes. Searching across position and scale lets features select a characteristic scale.
- SIFT aligns neighborhoods by scale and orientation, then summarizes local gradient directions in a normalized 128-component descriptor.
- SURF summarizes filter responses, ORB uses binary intensity comparisons, and shape context describes the arrangement of contour points.
- Nearest-neighbor matching finds the closest descriptor. Lowe's ratio test checks distinctiveness, while bidirectional matching checks whether both features choose each other.



Similar Descriptors Do Not Guarantee a Correct Match

Repeated patterns and missing counterparts can produce incorrect matches. Choose thresholds for the application and verify tentative correspondences.