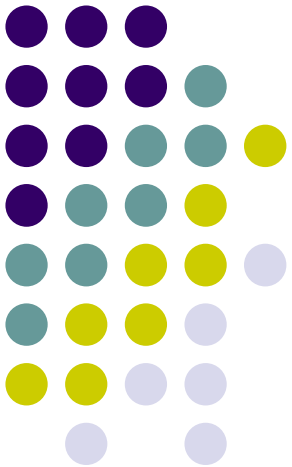
Digital Image Processing (CS/ECE 545)
Lecture 5: Edge Detection (Part 2)
& Corner Detection

Prof Emmanuel Agu

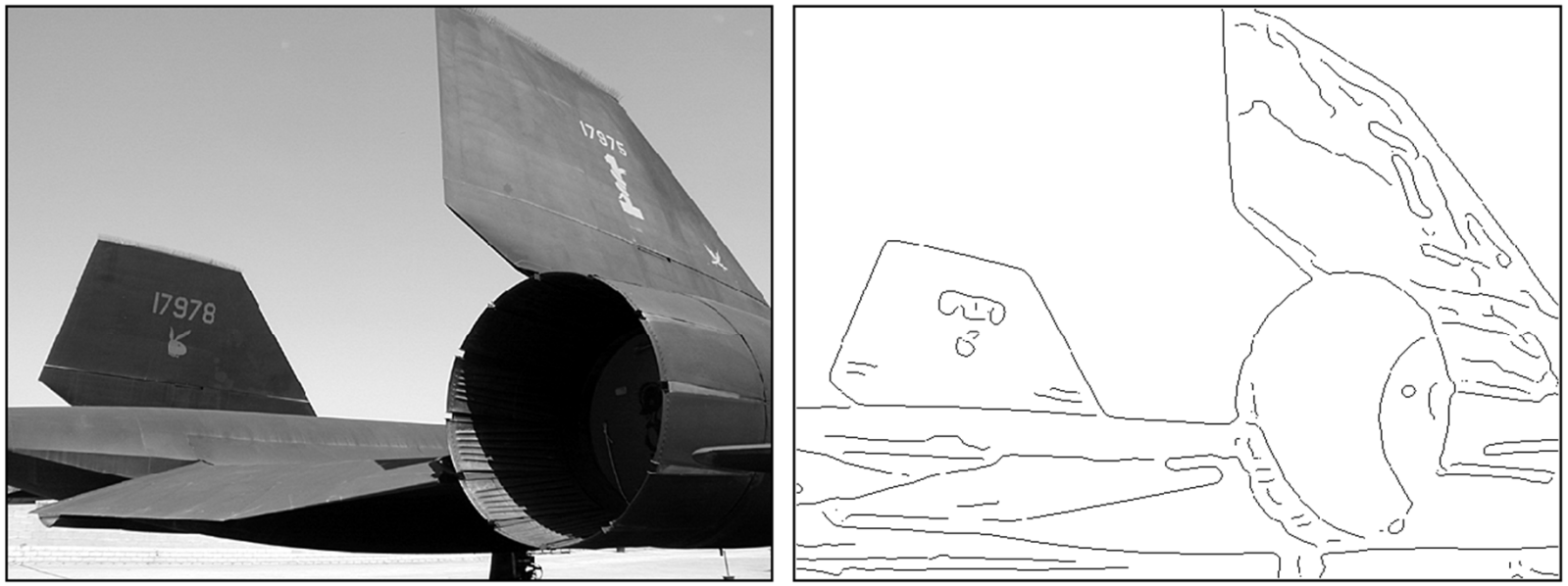
Computer Science Dept.

Worcester Polytechnic Institute (WPI)



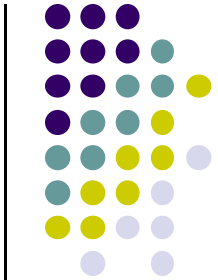
Recall: Edge Detection

- Image processing task that finds edges and contours in images
- Edges so important that human vision can reconstruct edge lines



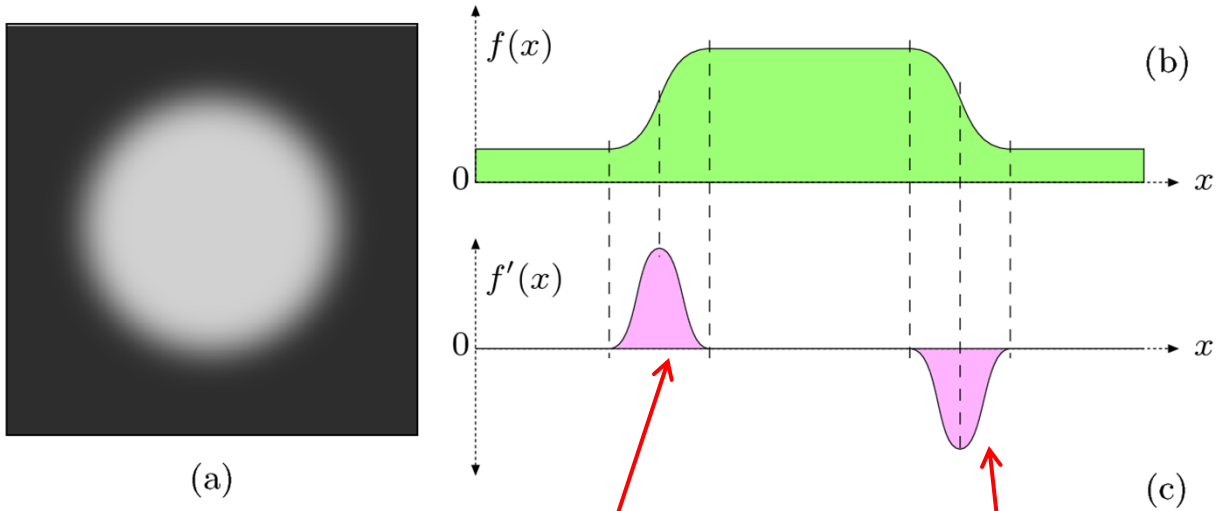
(a) (b)

Recall: Characteristics of an Edge



- Real (non-ideal) edge is a slightly blurred step function
- Edges can be characterized by high value first derivative

$$f'(x) = \frac{df}{dx}(x)$$



Rising slope causes positive + high value first derivative

Falling slope causes negative + high value first derivative

Recall: Image Gradient

- Image is 2D discrete function
- Image derivatives in horizontal and vertical directions

$$\frac{\partial I}{\partial u}(u,v) \quad \text{and} \quad \frac{\partial I}{\partial v}(u,v)$$

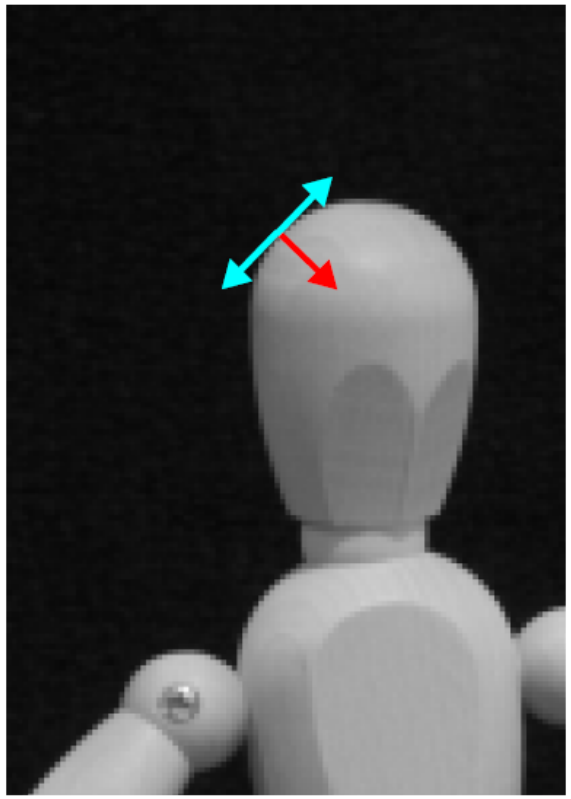
- Image gradient at location (u,v)

$$\nabla I(u,v) = \begin{bmatrix} \frac{\partial I}{\partial u}(u,v) \\ \frac{\partial I}{\partial v}(u,v) \end{bmatrix}$$

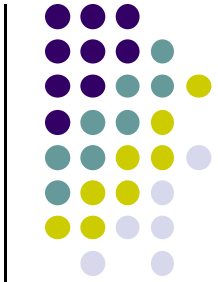
- Gradient magnitude

$$|\nabla I|(u,v) = \sqrt{\left(\frac{\partial I}{\partial u}(u,v)\right)^2 + \left(\frac{\partial I}{\partial v}(u,v)\right)^2}$$

- Magnitude is invariant under image rotation, used in edge detection



Recall: Gradient-Based Edge Detection



- Compute image derivatives by convolution

$$D_x(u,v) = H_x * I \quad \text{and} \quad D_y(u,v) = H_y * I$$

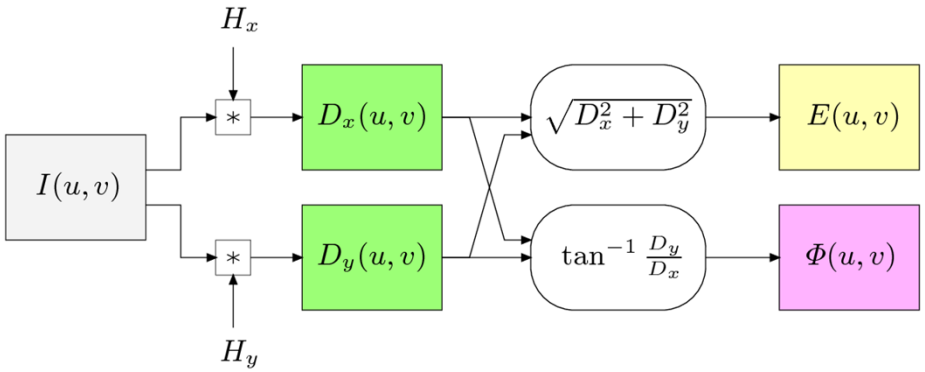
Scaled Filter results

- Compute edge gradient magnitude

$$E(u,v) = \sqrt{\big(D_x(u,v)\big)^2 + \big(D_y(u,v)\big)^2}$$

- Compute edge gradient direction

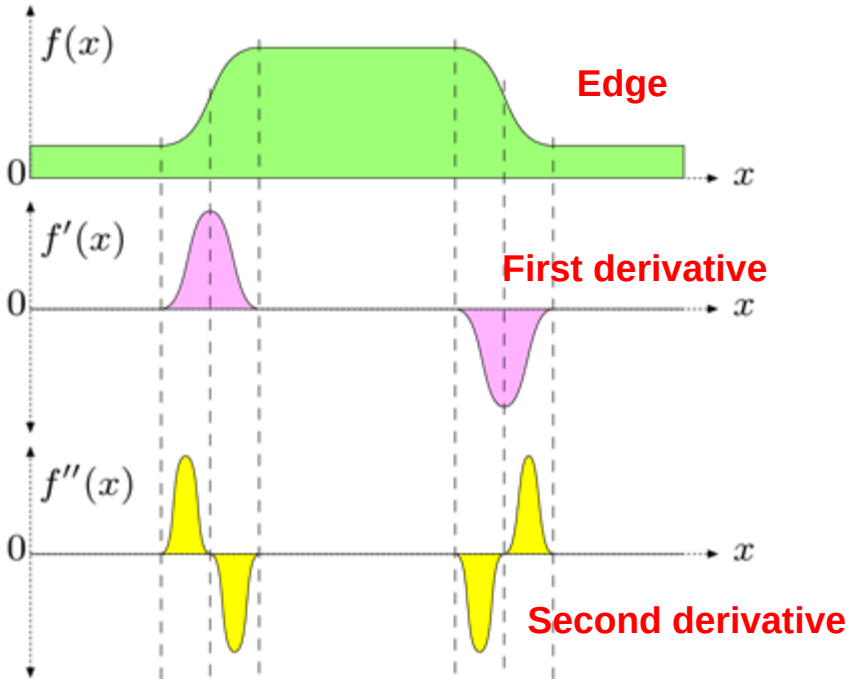
$$\Phi(u,v) = \tan^{-1}\left(\frac{D_y(u,v)}{D_x(u,v)}\right) = \mathrm{ArcTan}\big(D_x(u,v), D_y(u,v)\big)$$



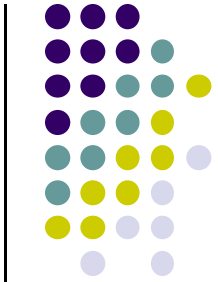
Typical process of Gradient based edge detection

Other Edge Operators

- **Problem with edge operators based on first derivatives:**
  - Edge is proportional to underlying intensity transition
  - Edges may be difficult to localize precisely
- Solution? Use second derivative
- **Recall:** An edge corresponds to a zero crossing of the 2nd derivative
- Since 2nd derivatives amplify image noise, pre-smoothing filters used first

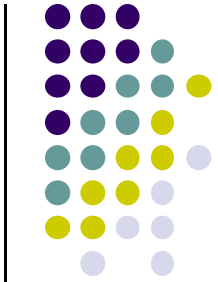


Edges at Different Scales



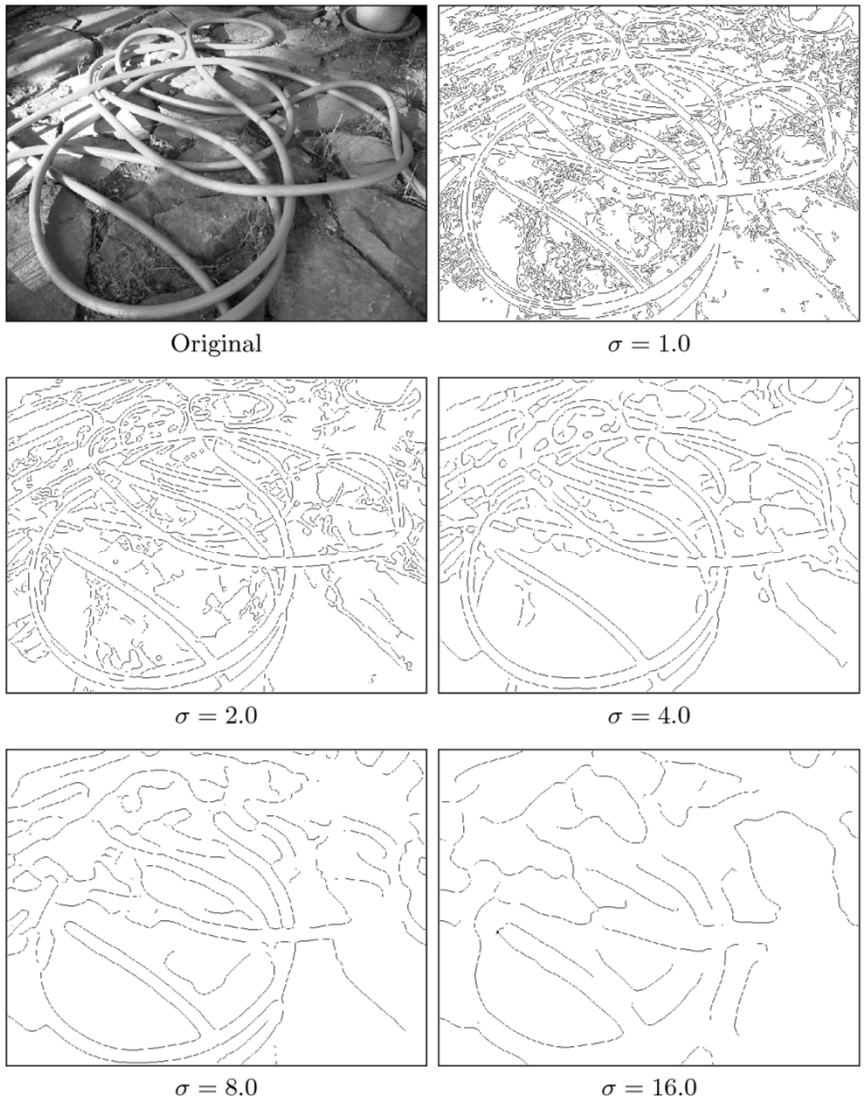
- Simple edge operators deviate from human perception in 2 main ways:
  - Edge operators respond to local intensity differences while human visual system **extends** edges across areas of minimal or vanishing contrast
  - Edges exist at multiple scales
- **Hierarchical or pyramid techniques:**
  - For each image position (u,v), apply edge detection filters at multiple scales
  - Use most dominant edge/scale detected (if any)

Canny Edge Detector



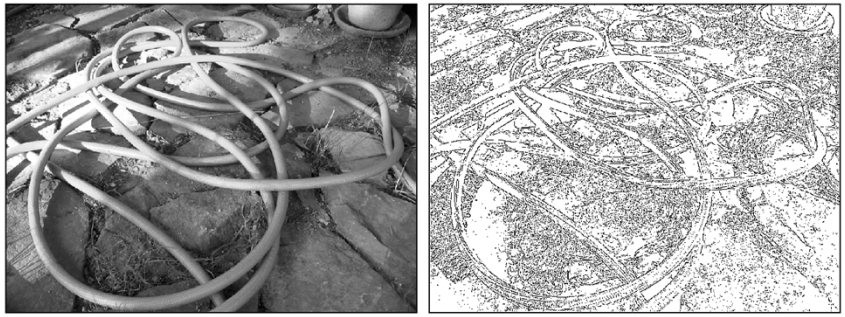
- Popular edge detector that operates at different scales, then combines results into common edge map. Tries to:
  1. Minimize number of false edge points
  2. Achieve good localization of edges
  3. Deliver only a single mark on each edge
- Essentially gradient based using zero crossings of second derivative
- Typically, a single scale implementation (1 image) used with adjustable filter radius (smoothing parameter σ)

Canny Edge Detector



Resulting edge maps for different settings of the smoothing (scale) parameter σ

Comparison of Various Edge Operators



Original

Roberts

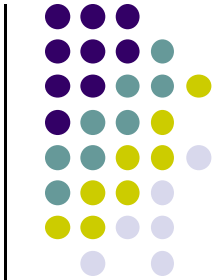
Prewitt

Sobel

Laplacian of Gaussian

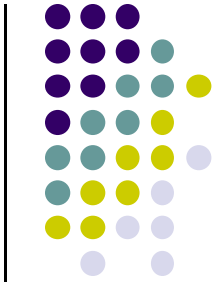
Canny ($\sigma = 1.0$)

From Edges to Contours



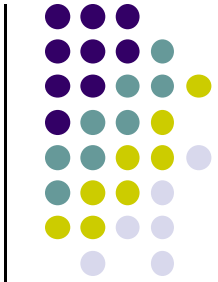
- Edge detection yields at each image position (u,v)
  - Edge strength + orientation
- How can this information be used to detect larger image structures and contours?
- **Contour following:** Starting from image point with high edge strength, follow edge iteratively till the 2 traces meet and a closed contour is formed
- **Several obstacles make this impractical and rarely used:**
  - Edges may end in regions of vanishing intensity gradient
  - Crossing edges lead to ambiguities
  - Contours may branch into several directions

Edge Maps



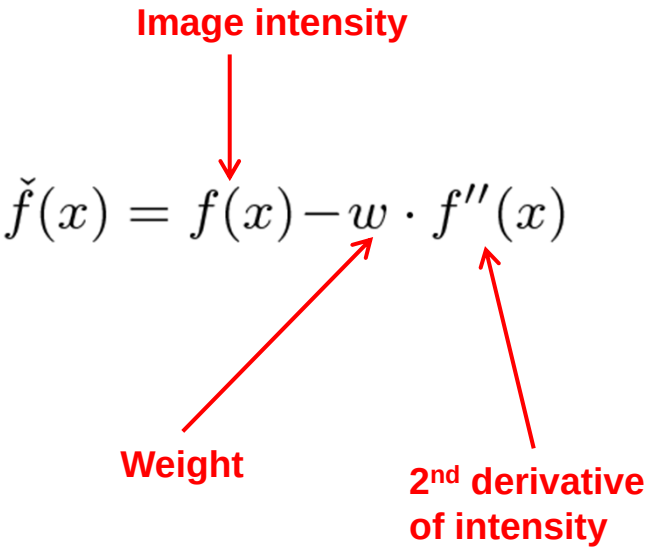
- Usually after calculating edge strengths, we just make binary decision whether point is valid edge
- **Most common approach:** generate **edge map** by appying threshold (fixed or adaptive) to edge strengths calculated by edge operator
- In practice edge maps seldom contain perfect contours
- Edge map frequently has small, unconnected contour fragments interrupted by positions of insufficient strength

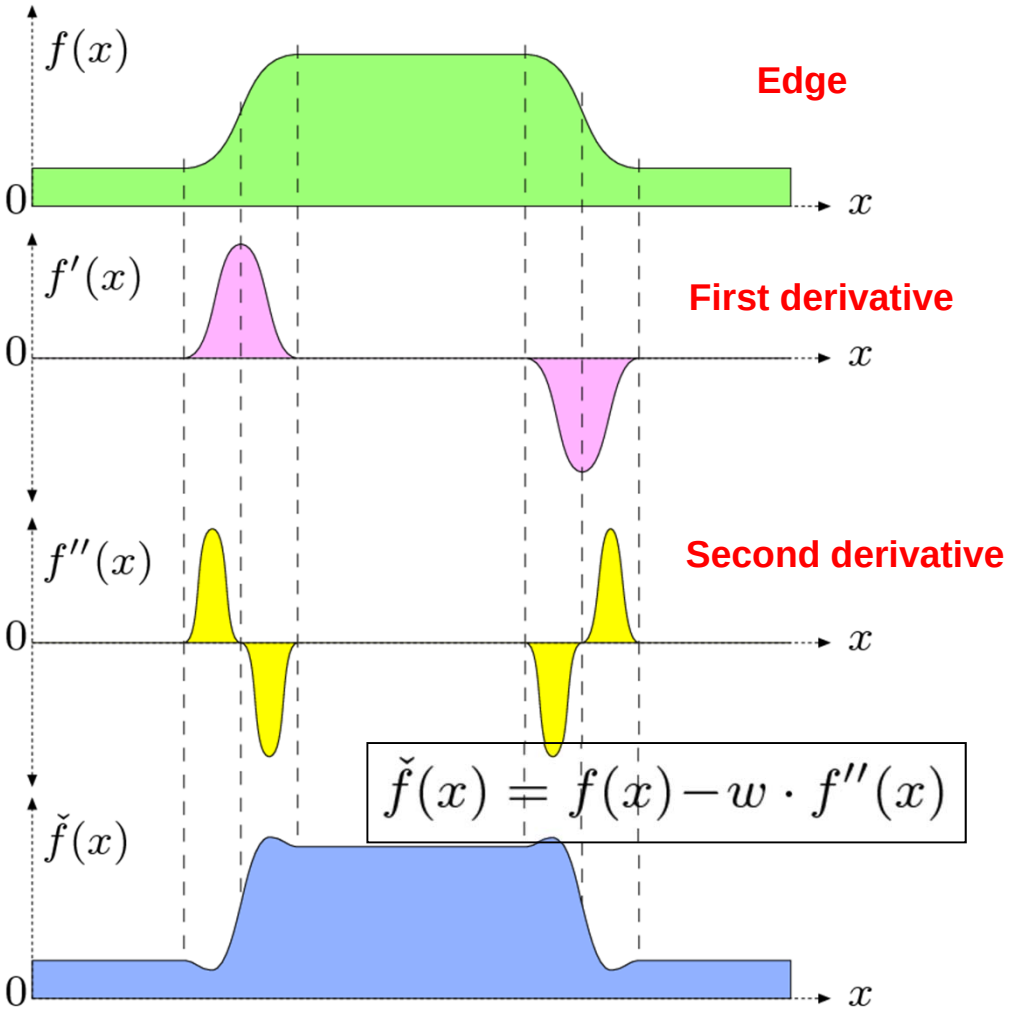
Image Sharpening



- Blurring may occur during image scanning or scaling
- Sharpening reduces effects of blurring
- How? Amplify high frequency components
- High frequencies occur at edges
- We need to sharpen edges
- Two main approaches:
  - Using Laplace filter
  - Unsharp masking

Edge Sharpening using Laplace Filter





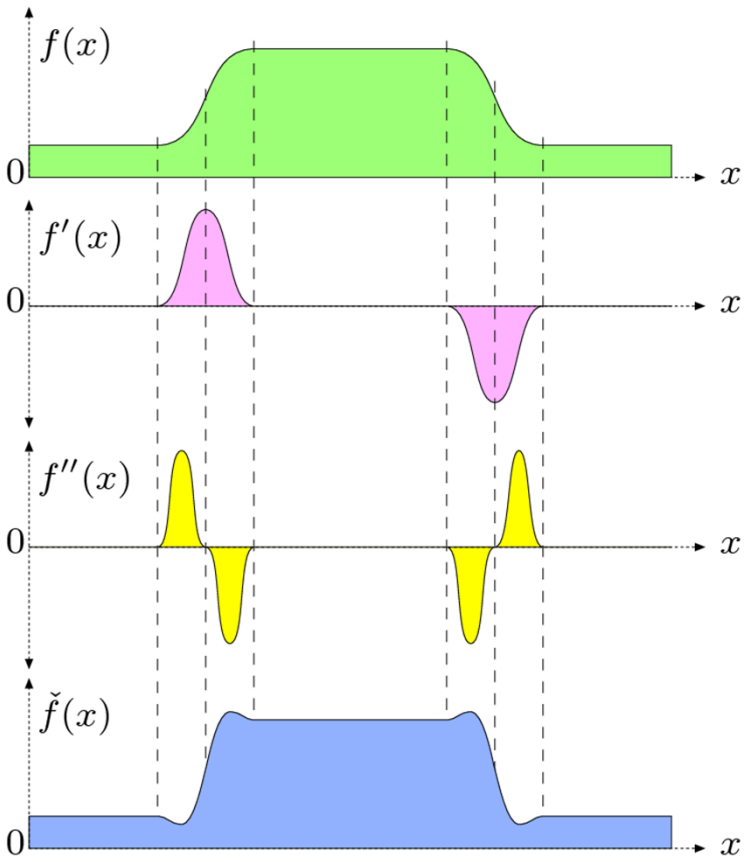
Laplace Operator

- **2D Laplace operator:** combines 2nd derivatives in horizontal and vertical directions
- Laplace operator defined as:

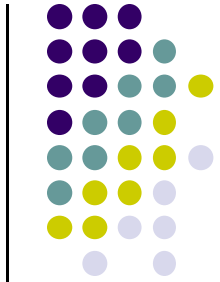
$$(\nabla^2 f)(x,y) = \frac{\partial^2 f}{\partial^2 x}(x,y) + \frac{\partial^2 f}{\partial^2 y}(x,y)$$

2nd derivative of intensity in x direction

2nd derivative of intensity in y direction



Laplacian Operator



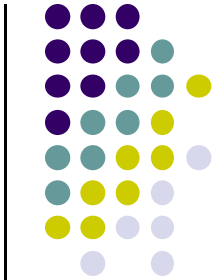
- Laplacian: $(\nabla^2 f)(x,y) = \frac{\partial^2 f}{\partial^2 x}(x,y) + \frac{\partial^2 f}{\partial^2 y}(x,y)$
- Digital approximation of laplacian is:

$$\nabla^2 f(x,y) = [f(x+1,y) - f(x,y)] - [f(x,y) - f(x-1,y)] + [f(x,y+1) - f(x,y)] - [f(x,y) - f(x,y-1)]$$

$$= [f(x+1,y) + f(x-1,y) + f(x,y+1) - f(x,y-1)] - 4f(x,y)$$

0	1	0
1	−4	1
0	1	0

Laplacian Operator



- Laplacian: $(\nabla^2 f)(x,y) = \frac{\partial^2 f}{\partial^2 x}(x,y) + \frac{\partial^2 f}{\partial^2 y}(x,y)$

1d filters that estimate 2nd derivatives along x and y directions

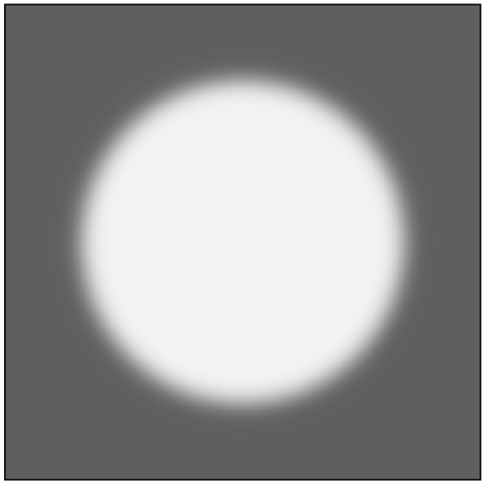
$$\frac{\partial^2 f}{\partial^2 x} \equiv H_x^L = \begin{bmatrix} 1 & -\mathbf{2} & 1 \end{bmatrix} \quad \text{and} \quad \frac{\partial^2 f}{\partial^2 y} \equiv H_y^L = \begin{bmatrix} 1 \\ -\mathbf{2} \\ 1 \end{bmatrix}$$

$$H^L = H_x^L + H_y^L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -\mathbf{4} & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

2-dimensional Laplace filter

Results for Laplacian Operator

Synthetic test Image *I*

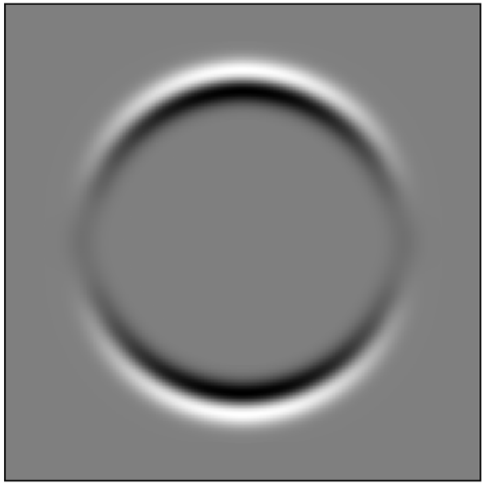


(a)

(b)

Second partial derivative in horizontal direction

Second partial derivative in Vertical direction



(c)

(d)

Laplace filter

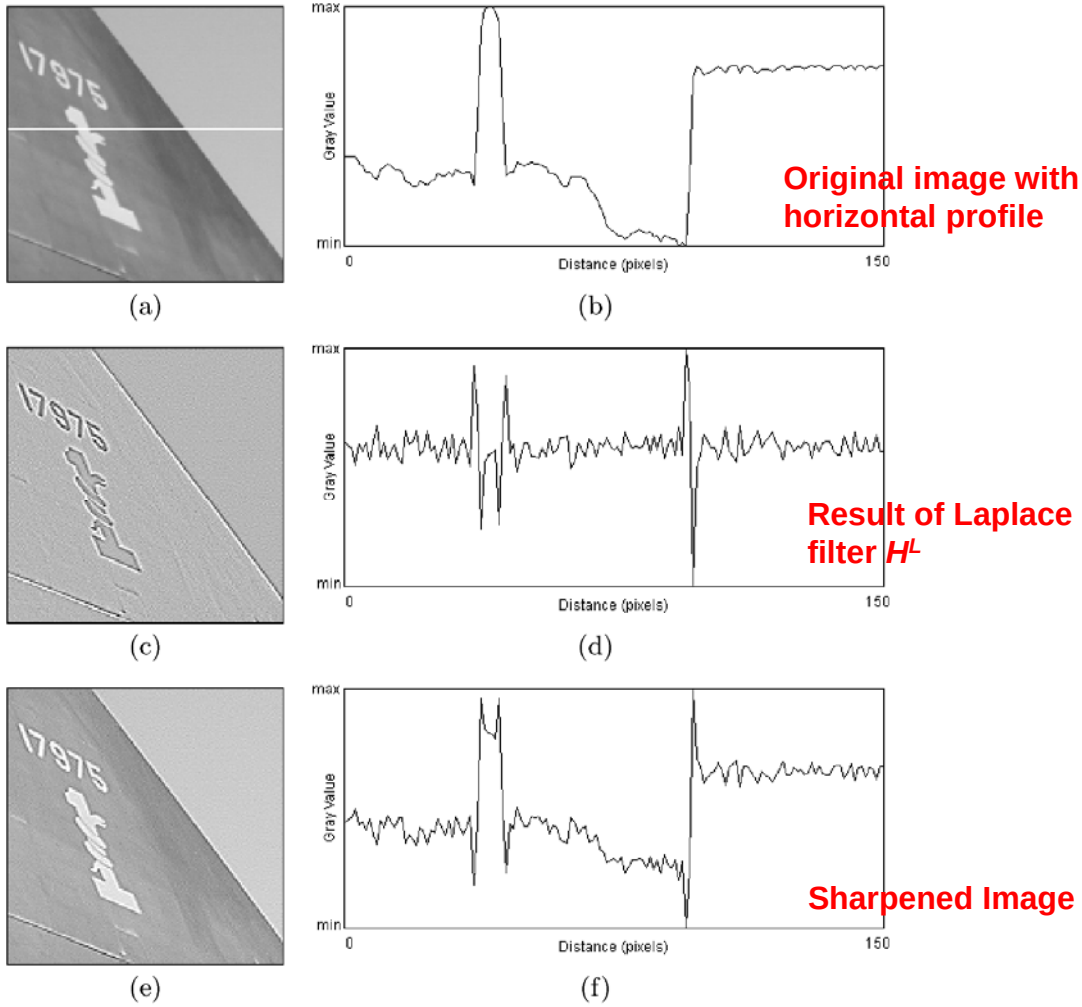
Edge Sharpening

- To perform actual sharpening:
  - First apply Laplace filter to Image *I*
  - Subtract fraction of result from original image

$$\check{I} \leftarrow I - w \cdot (H^L * I)$$

$$H^L = H_x^L + H_y^L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

2-dimensional Laplace filter

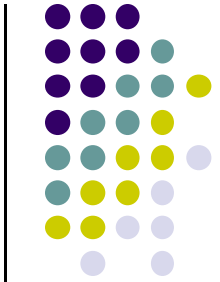


Original image with horizontal profile

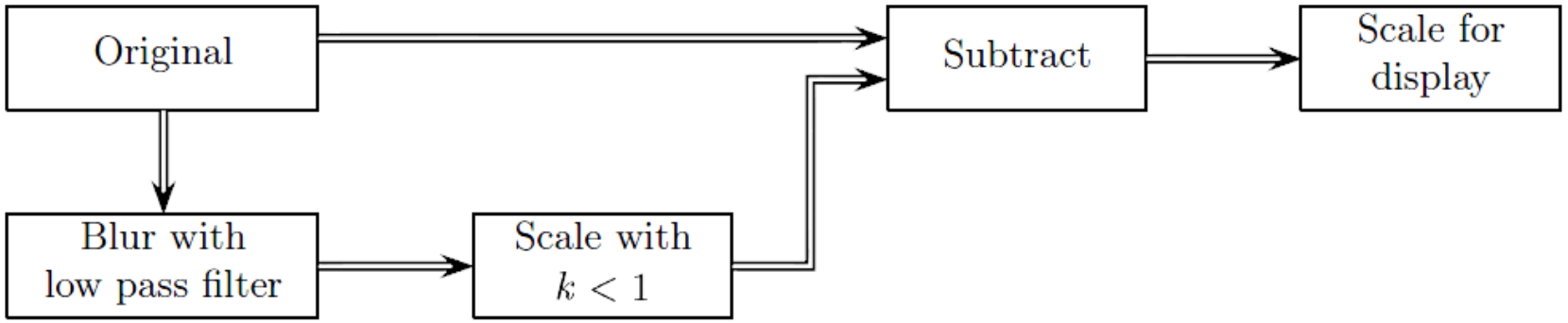
Result of Laplace filter H^L

Sharpened Image

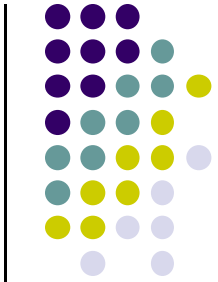
Unsharp Masking (USM)



- A technique for edge sharpening
  - Combine image with smoothed (blurred) version of image



Unsharp Masking (USM) Steps



- Subtract smooth version (Gaussian smoothing) from image to obtain **enhanced edge mask**

$$M \leftarrow I - (I * \tilde{H}) = I - \tilde{I}$$

- Add the mask to image with a weight

$$\check{I} \leftarrow I + a \cdot M$$

- Together:

$$\check{I} \leftarrow I + a \cdot (I - \tilde{I}) = (1 + a) \cdot I - a \cdot \tilde{I}$$

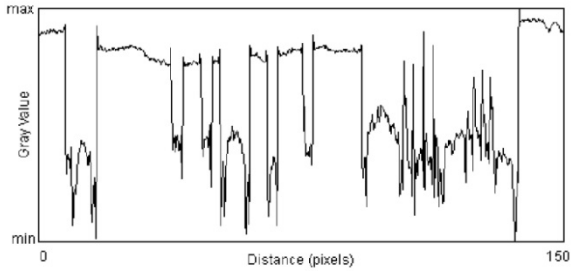
- Advantages of USM over Laplace filter
  - Reduced noise sensitivity due to smoothing
  - Improved control through parameters σ and a

Unsharp Masking (USM)

original

(a) Original

(b)



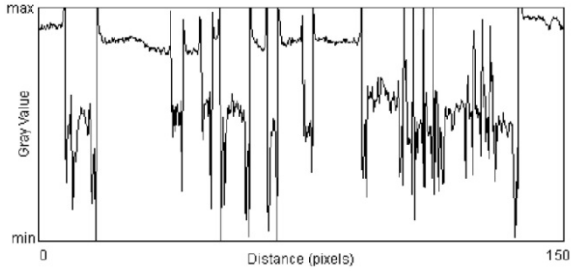
(c)

Intensity profile at line

Results of USM (σ = 2.5)

(d) $\sigma = 2.5$

(e)

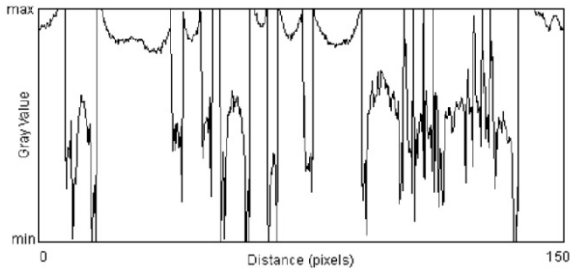


(f)

Results of USM (σ = 10.0)

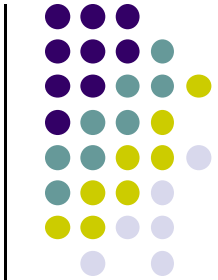
(g) $\sigma = 10.0$

(h)



(i)

Unsharp Masking Implementation



```
1   public void unsharpMask(ImageProcessor ip,
2         double sigma, double a) {
3     ImageProcessor I = ip.convertToFloat(); // I
4
5     // create a blurred version of the image
6     ImageProcessor J = I.duplicate();       // Ĩ
7     float[] H = GaussKernel1d.create(sigma); // see Prog. 6.4
8     Convolver cv = new Convolver();
9     cv.setNormalize(true);
10    cv.convolve(J, H, 1, H.length);
11    cv.convolve(J, H, H.length, 1);
12
13    I.multiply(1+a);     // I ← (1 + a) · I
14    J.multiply(a);       // Ĩ ← a · Ĩ
15    I.copyBits(J,0,0,Blitter.SUBTRACT); // Ĩ ← (1 + a) · I − a · Ĩ
16
17    //copy result back into original byte image
18    ip.insert(I.convertToByte(false), 0, 0);
19  }
```

Convert original image to FloatProcessor object

Create image to hold blurred copy

Create gaussian kernel

Apply gaussian kernel in Vertical and horizontal directions

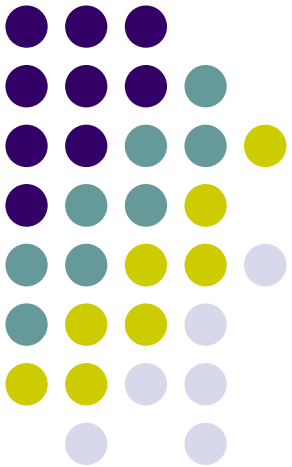
USM implemented in ImageJ by plugin class `ij.plugin.filter.UnsharpMask`

Digital Image Processing (CS/ECE 545)
Lecture 5: Edge Detection (Part 2)
& Corner Detection

Prof Emmanuel Agu

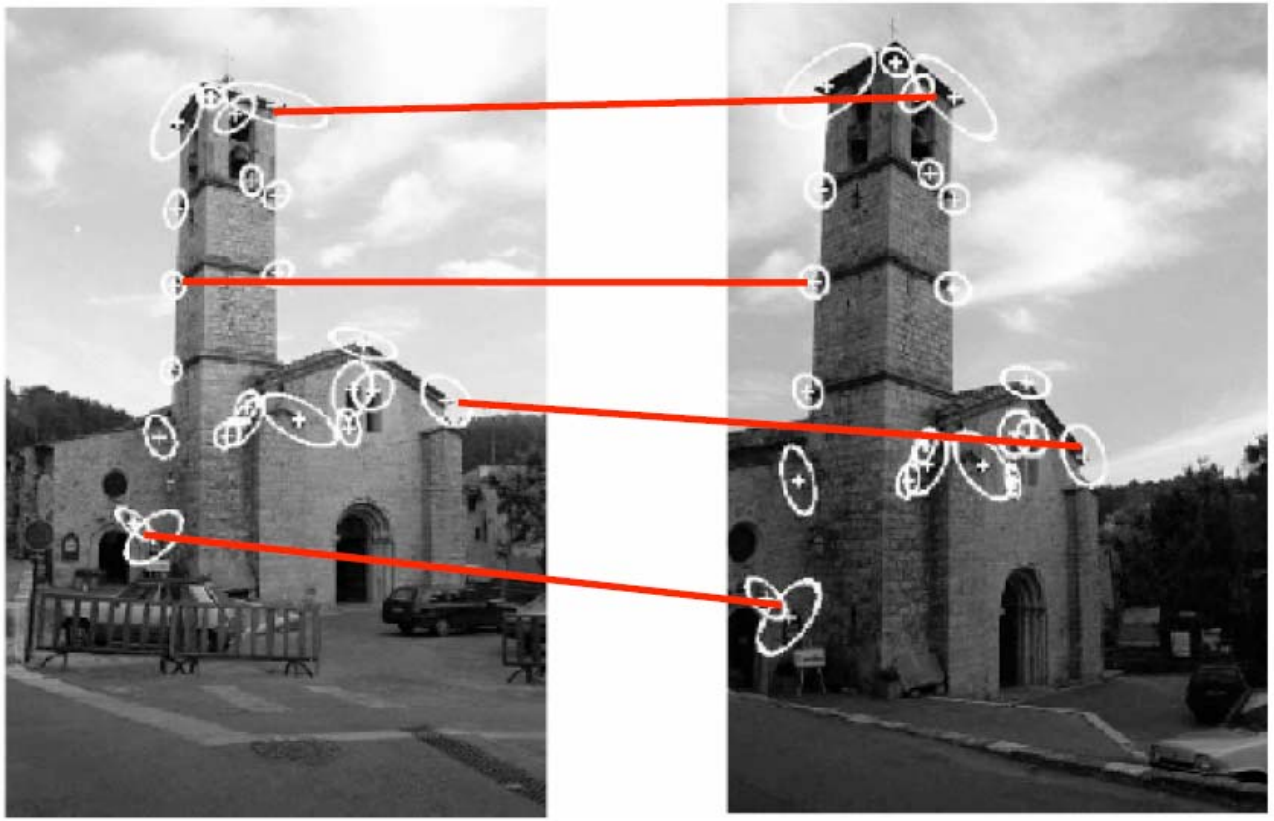
Computer Science Dept.

Worcester Polytechnic Institute (WPI)

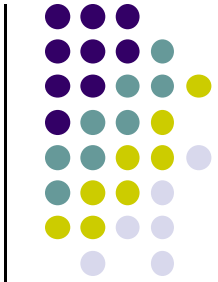


Why Corner Detection?

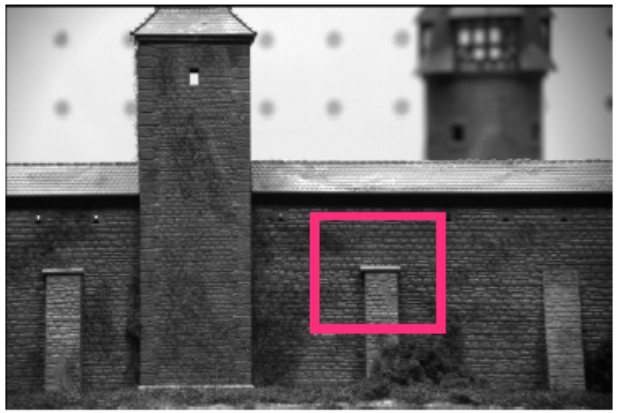
- Corners are robust, used in computer vision for matching same point in multiple images (e.g. stereo left + right image)

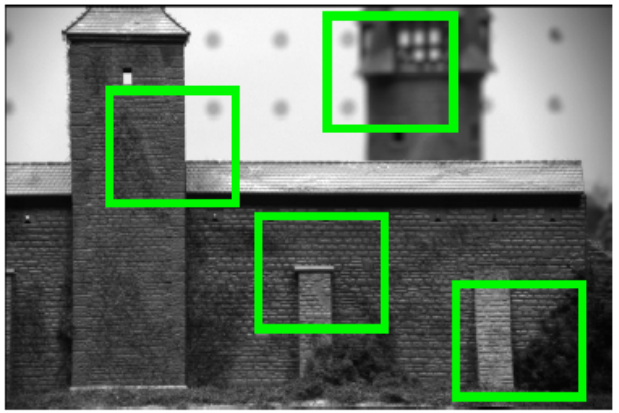


Motivation: Patch Matching

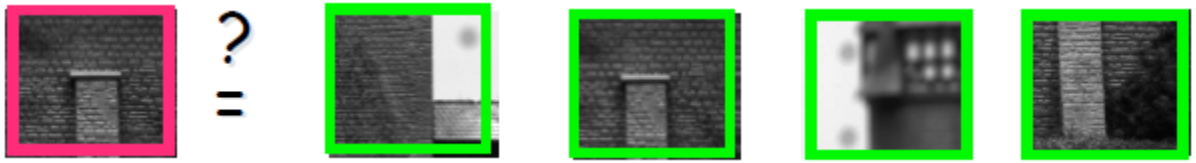


- Consider small squares called **patches** of same size in both images

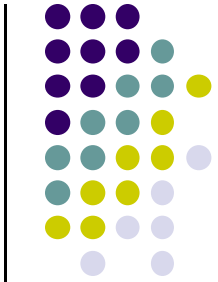




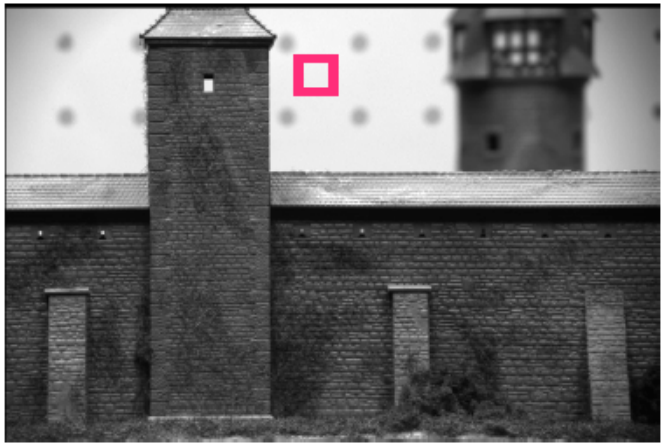
- **The task?** Find most similar patch in second image

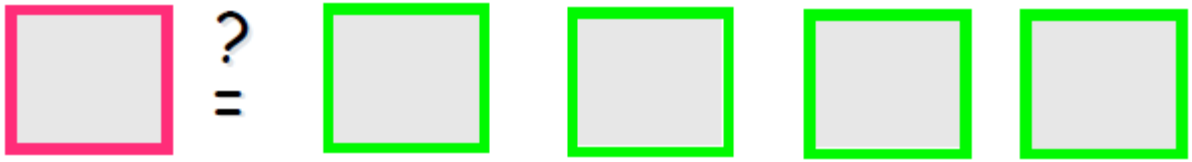


Some Patches Better Than Others

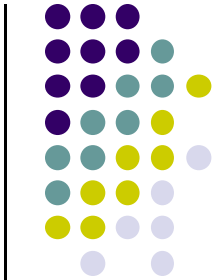


- More robust to use **distinctive patches**
- Example: Don't use patches similar to many patches in image 2 (ambiguous)

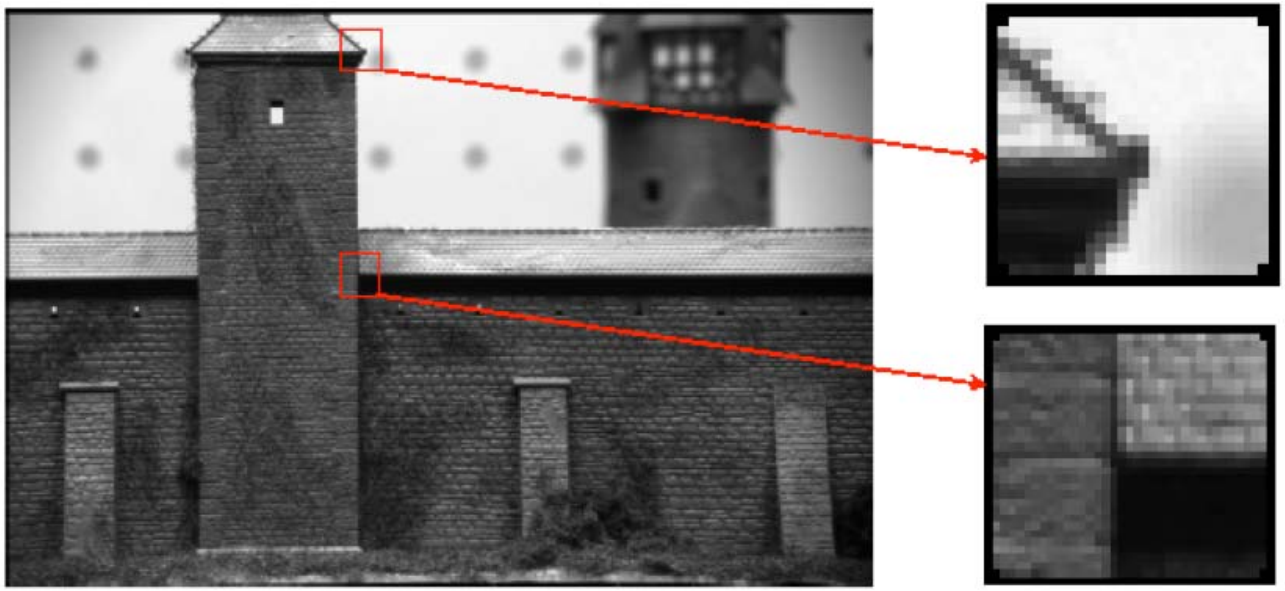




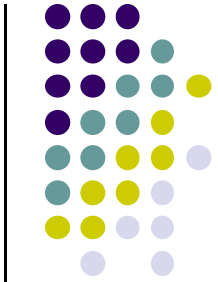
Corners are Robust “Features”



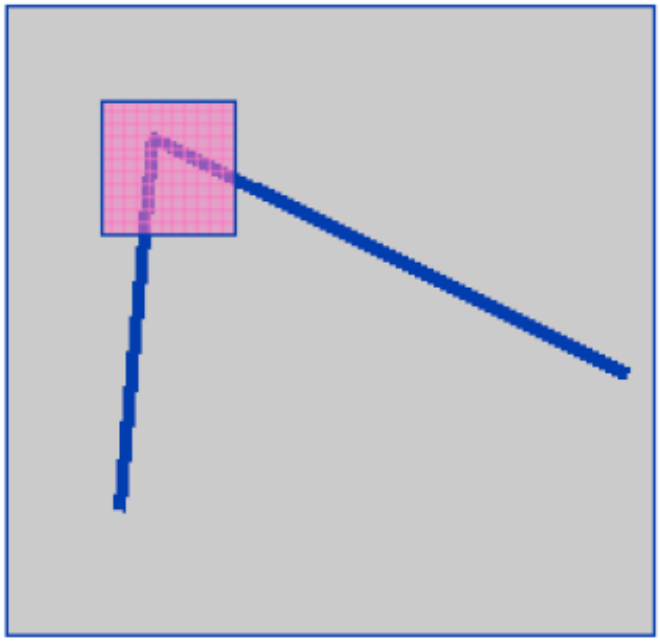
- Corners are unique, match patches with corners
- What are corners? Junctions of contours
- Corners appear as large changes in intensity in different viewpoints (stable/unique)



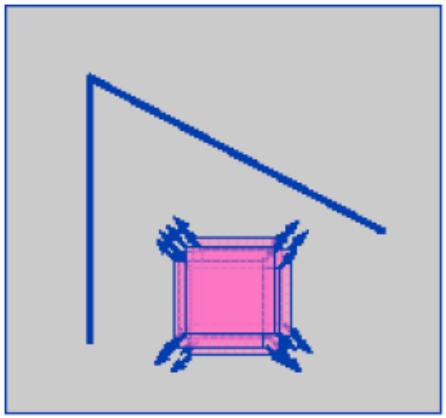
Corner Points: Basic Idea



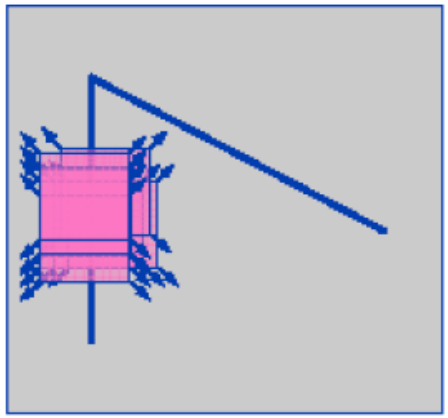
- **Patch at corner:** shifting window in any direction yields large change in intensity
- **Is patch at corner?** shift window in multiple directions, if large intensity changes, patch is a corner



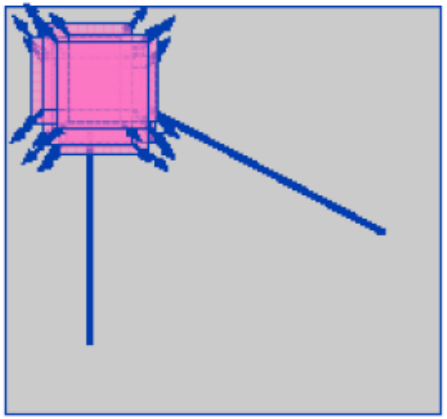
Harris Corner Detector: Basic Idea



Flat region: no intensity change in all directions



Edge: no change along edge directions

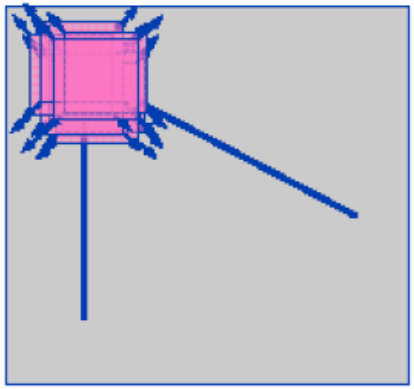


Corner: significant intensity change in many directions

- Harris corner detector gives mathematical approach for determining which case holds

Harris Detector: The Mathematics

- Shift patch by [*u,v*] and compute change in intensity
- Change of intensity for shift [*u,v*]



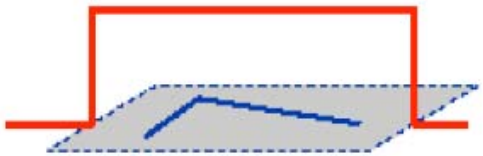
$$E(u,v) = \sum_{x,y} w(x,y)\left[I(x+u,y+v) - I(x,y)\right]^2$$

Window function

Shifted intensity

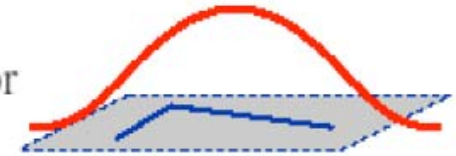
Intensity

Window function $w(x,y)$ =



1 in window, 0 outside

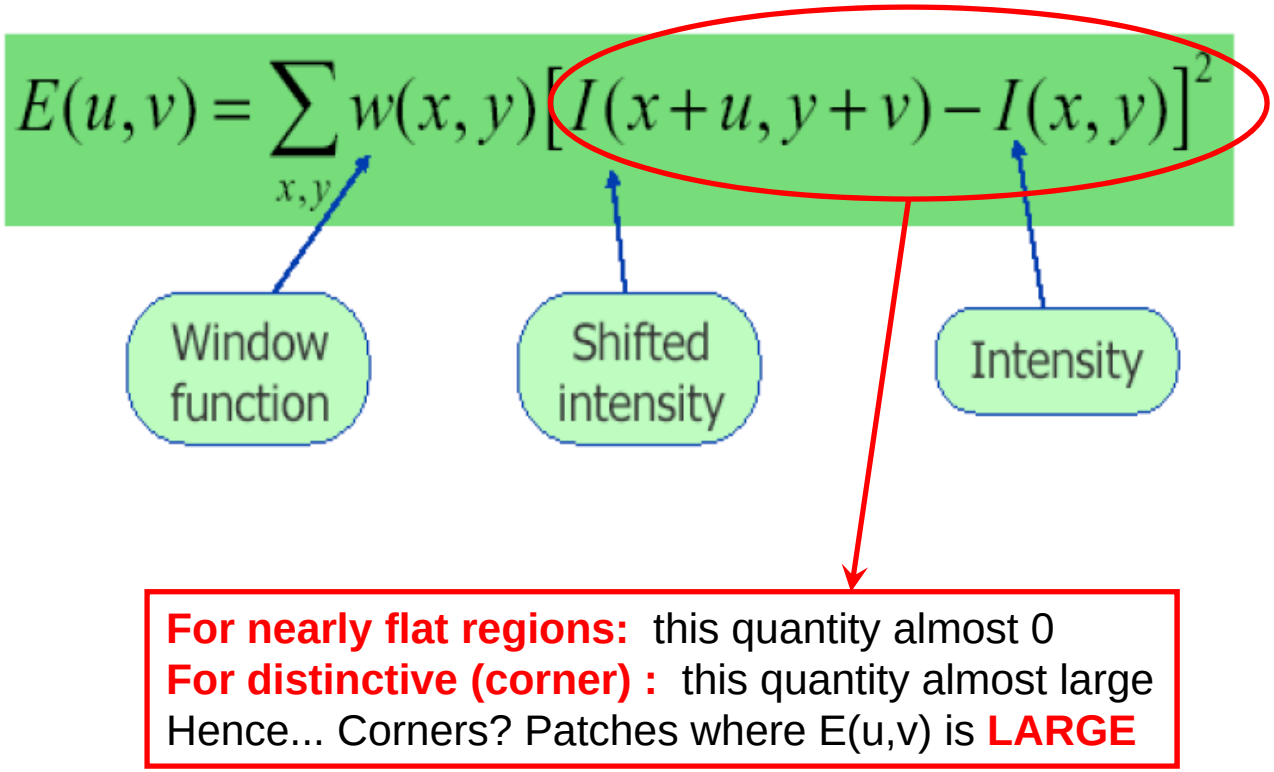
or



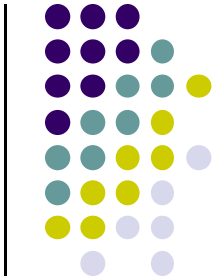
Gaussian

Harris Detector: The Intuition

- Change of intensity for shift [*u,v*]



Taylor Series Expansion



$$f(x+u,y+v) = f(x,y) + uf_x(x,y) + vf_y(x,y) +$$

First partial derivatives

$$\frac{1}{2!}\left[u^2 f_{xx}(x,y) + uvf_{xy}x,y + v^2 f_{yy}(x,y)\right] +$$

Second partial derivatives

$$\frac{1}{3!}\left[u^3 f_{xxx}(x,y) + u^2 v f_{xxy}(x,y) + uv^2 f_{xyy}(x,y) + v^3 f_{yyy}(x,y)\right]$$

Third partial derivatives

$$+ \ldots \text{ (Higher order terms)}$$

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

Window function

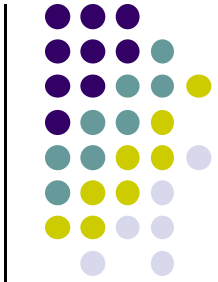
Shifted intensity

Intensity

$$\approx \sum \left[I(x,y) + uI_x + vI_y - I(x,y)\right]^2$$

First order approximation

Harris Corner Derivation



$$\sum [I(x+u,y+v) - I(x,y)]^2$$

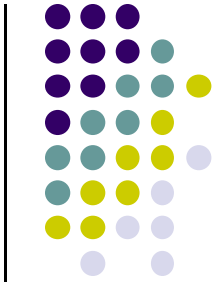
$$\approx \sum [I(x,y) + uI_x + vI_y - I(x,y)]^2 \qquad \text{First order approx}$$

$$= \sum u^2 I_x^2 + 2uvI_xI_y + v^2 I_y^2$$

$$= \sum \begin{bmatrix} u & v \end{bmatrix} \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \qquad \text{Rewrite as matrix equation}$$

$$= \begin{bmatrix} u & v \end{bmatrix} \left(\sum \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix} \right) \begin{bmatrix} u \\ v \end{bmatrix}$$

Harris Corner Detector



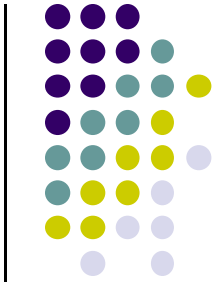
- For small shifts [*u, v*], we have following approximation

$$E(u,v) \cong [u,v] \quad M \quad \begin{bmatrix} u \\ v \end{bmatrix}$$

where *M* is the 2x2 matrix

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

Harris Corner Detector



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

- If we compute derivatives *A(u,v), B(u,v)* and *C(u,v)*

$$A(u,v) = I_x^2(u,v)$$

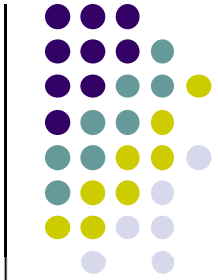
$$B(u,v) = I_y^2(u,v)$$

$$C(u,v) = I_x(u,v) \cdot I_y(u,v)$$

- Can express matrix *M* as

$$\boldsymbol{M} = \begin{pmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{pmatrix} = \begin{pmatrix} A & C \\ C & B \end{pmatrix}$$

Harris Corner Detector



- Smooth *A(u,v), B(u,v)* and *C(u,v)* individually with linear gaussian filter

$$\bar{M} = \begin{pmatrix} A*H^{G,\sigma} & C*H^{G,\sigma} \\ C*H^{G,\sigma} & B*H^{G,\sigma} \end{pmatrix} = \begin{pmatrix} \bar{A} & \bar{C} \\ \bar{C} & \bar{B} \end{pmatrix}$$

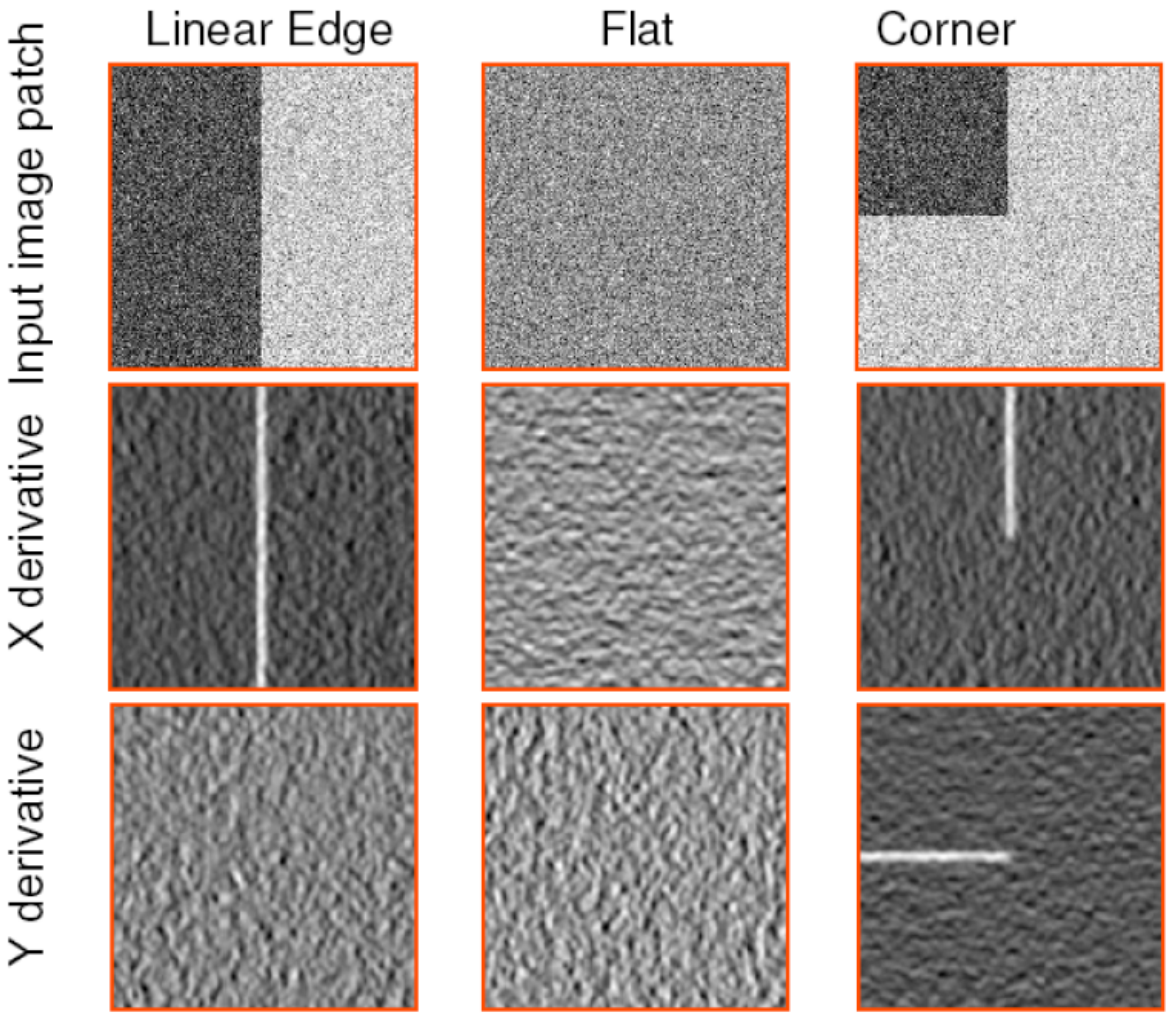
- Since the matrix $\bar{M}$ is symmetric, it can be diagonalized

$$\bar{M}' = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

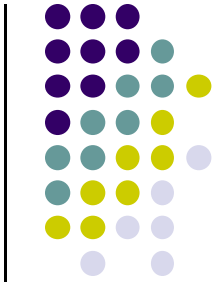
where λ_1, λ_2 are **eigenvalues** of matrix $\bar{M}$ defined as:

$$\lambda_{1,2} = \frac{\mathrm{trace}(\bar{M})}{2} \pm \sqrt{\left(\frac{\mathrm{trace}(\bar{M})}{2}\right)^2 - \det(\bar{M})}$$
$$= \frac{1}{2}\left(\bar{A} + \bar{B} \pm \sqrt{\bar{A}^2 - 2\bar{A}\bar{B} + \bar{B}^2 + 4\bar{C}^2}\right)$$

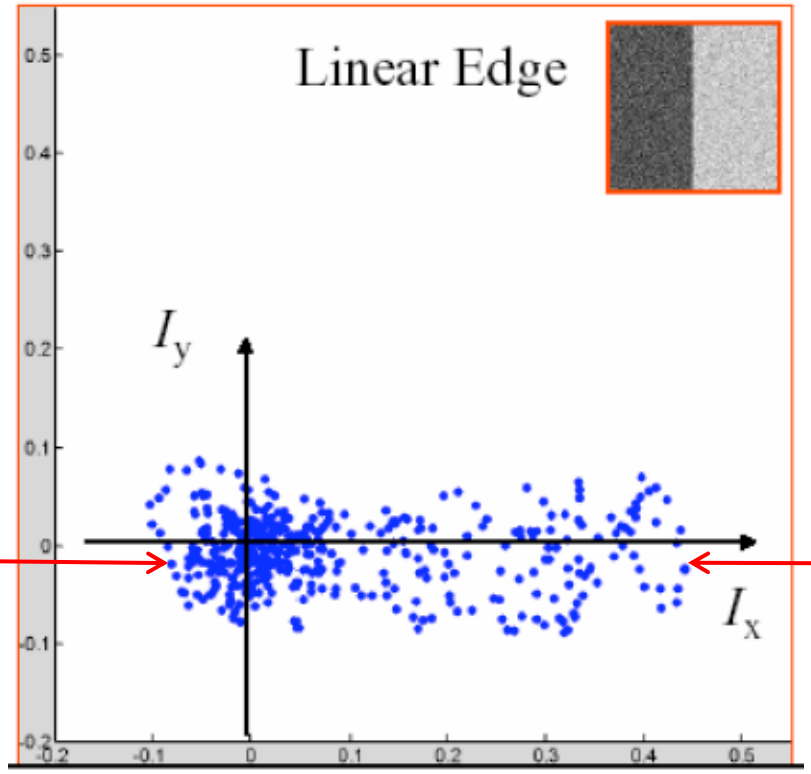
Harris Corner Detection Intuition
Example: Cases and 2D Derivatives



Some Intuition: Plotting Derivatives as 2D Points



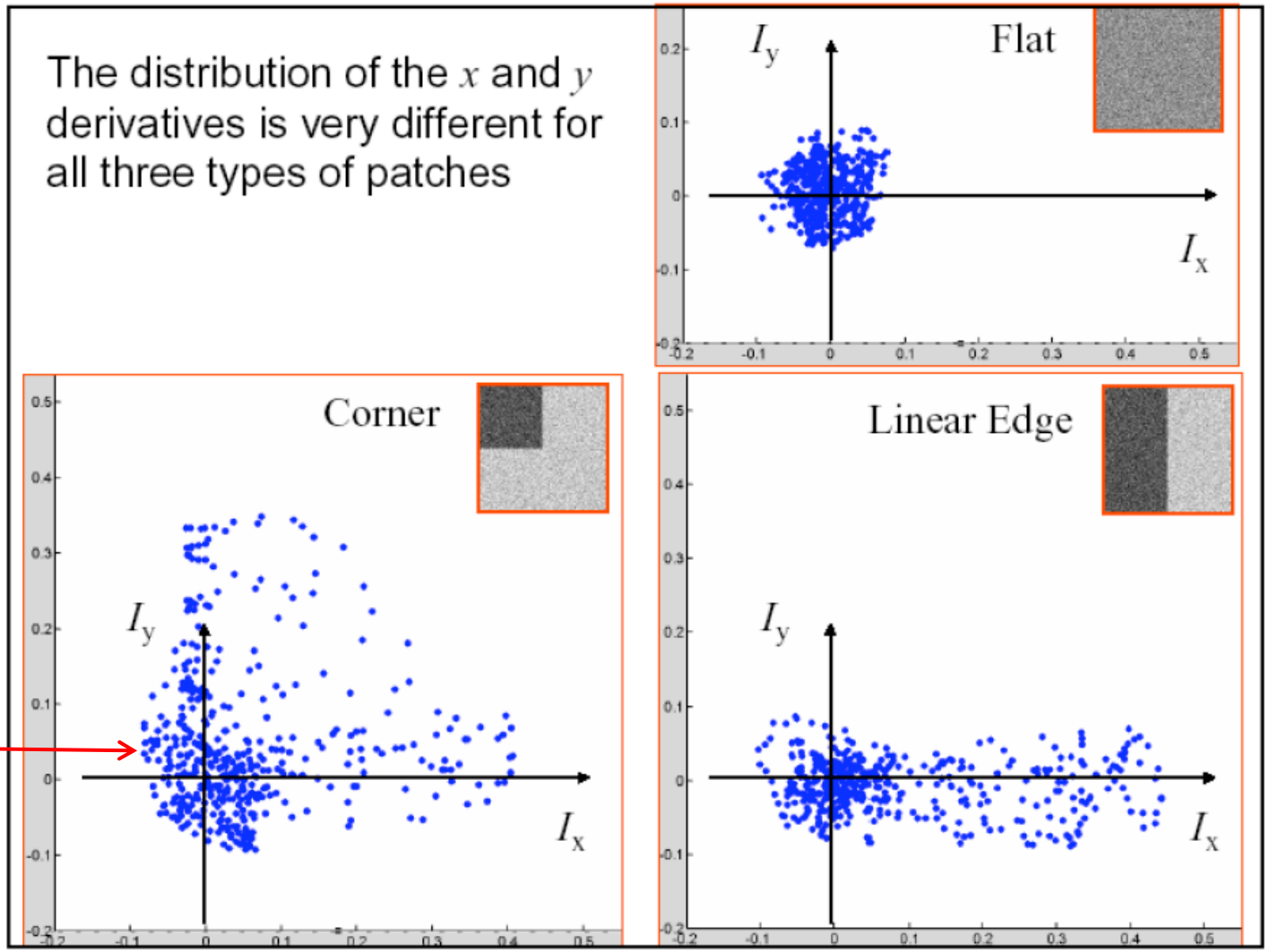
- Treat gradient vectors as set of (dx,dy) points with center at (0,0)
- Fit an ellipse to that set of points via scatter matrix
- Analyze ellipse parameters for varying cases…



Flat regions (equal dx,dy)

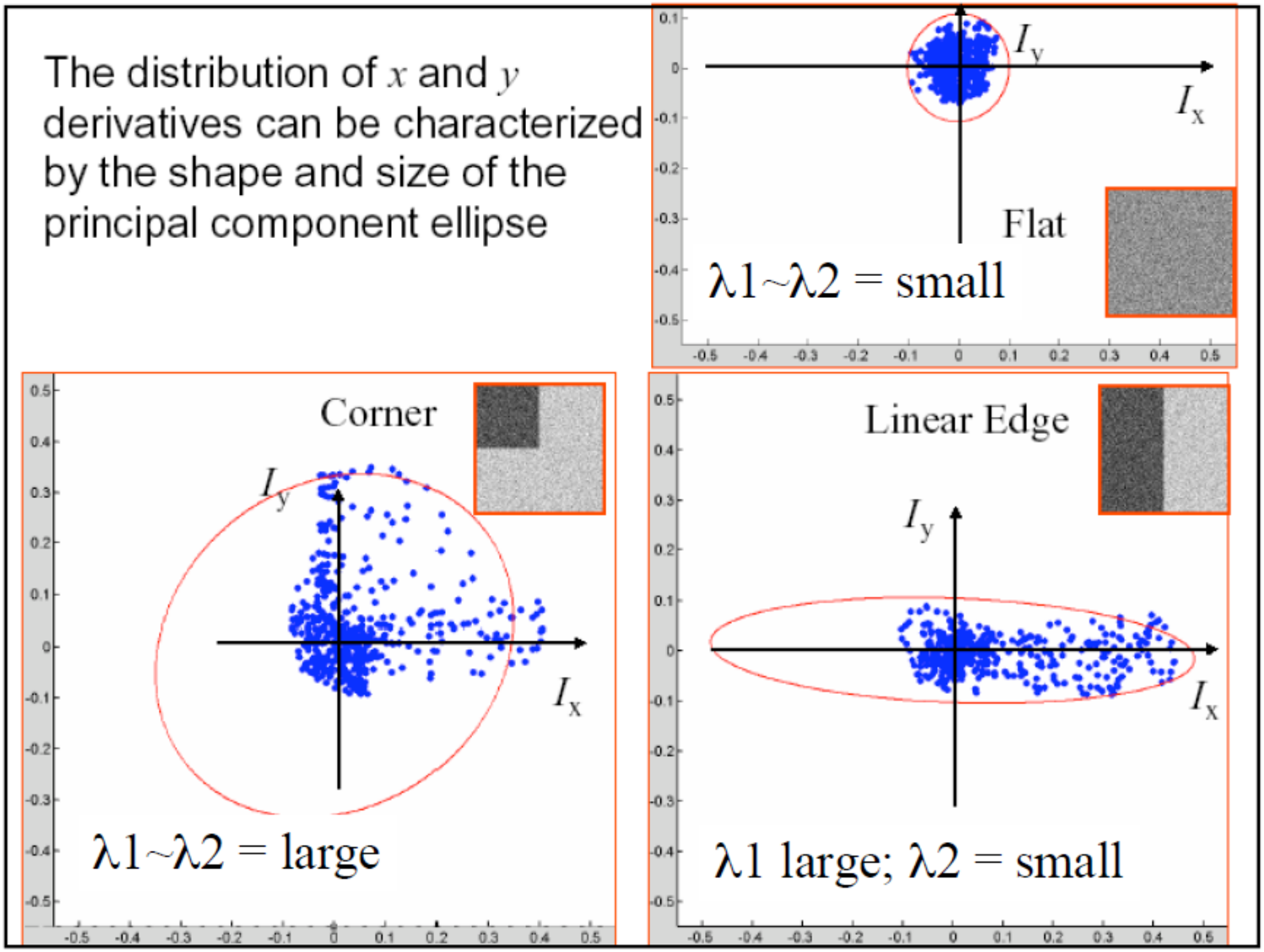
Edge Regions (large dx, small dy)

Plotting Derivatives as 2D Points

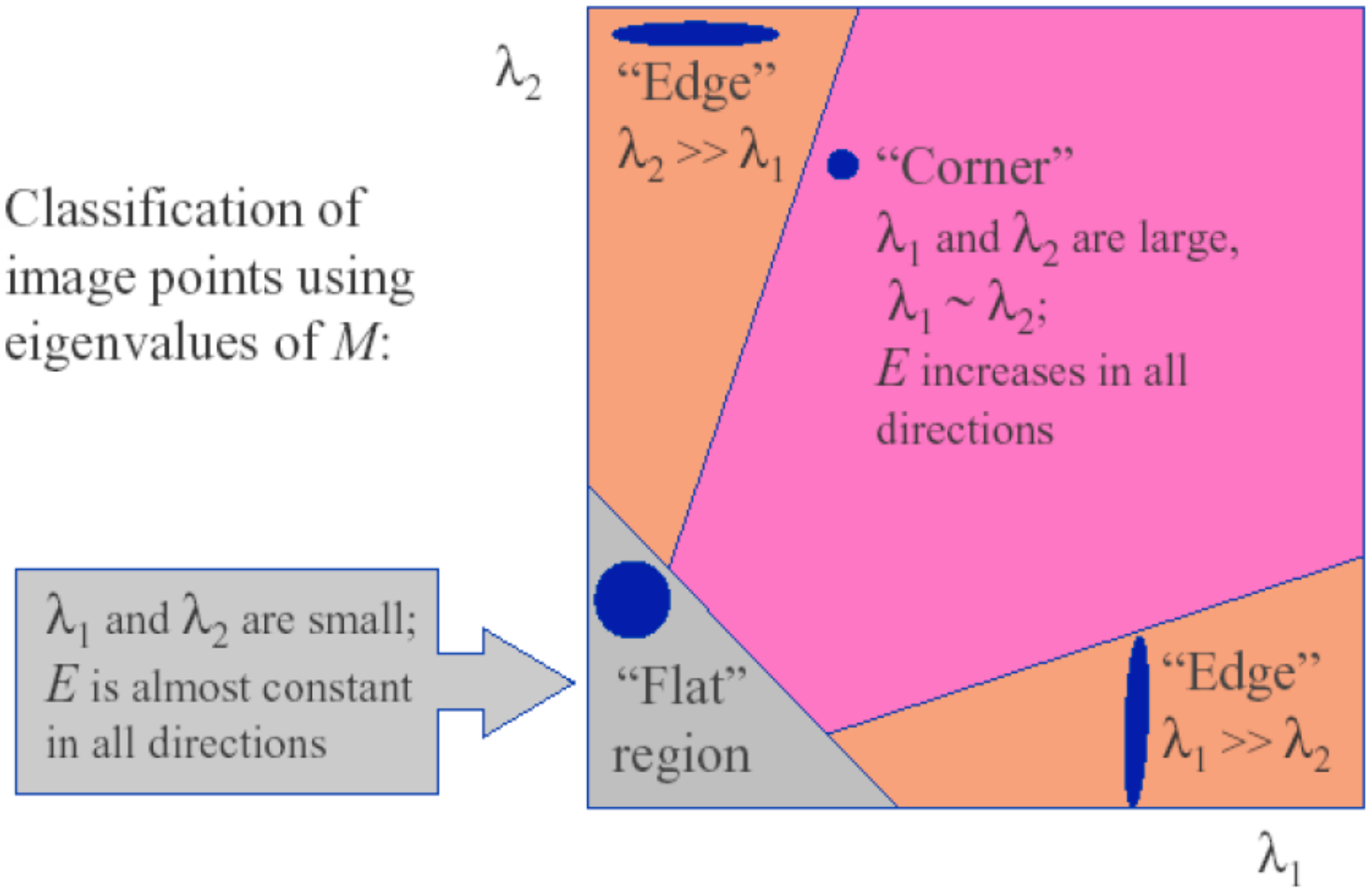


Corner (large dx,dy)

Fitting Ellipse to Each Set of Points



Classification of Eigenvalues

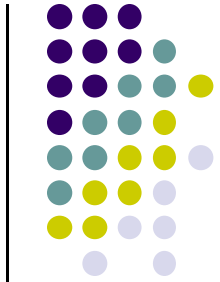


Harris Corner Detector

- $$\bar{M}' = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \qquad \lambda_{1,2} = \frac{\mathrm{trace}(\bar{M})}{2} \pm \sqrt{\left(\frac{\mathrm{trace}(\bar{M})}{2}\right)^2 - \det(\bar{M})}$$
$$= \frac{1}{2}\left(\bar{A} + \bar{B} \pm \sqrt{\bar{A}^2 - 2\bar{A}\bar{B} + \bar{B}^2 + 4\bar{C}^2}\right)$$

- Eigenvalues λ_1, λ_2 encode **edge strength**
  - Flat (uniform regions of image) $\bar{M}$ = 0. Therefore $\lambda_1 = \lambda_2 = 0$
  - For ideal ramp, $\lambda_1 > 0$, $\lambda_2 > 0$
- Associated eigenvectors represent **edge orientation**
- A corner should have:
  - Strong edge in main direction (corresponding to larger of λ_1, λ_2)
  - Another edge normal to the first (corresponding to smaller of λ_1, λ_2)

Harris Corner Detector



- $$\lambda_{1,2} = \frac{\text{trace}(\bar{M})}{2} \pm \sqrt{\left(\frac{\text{trace}(\bar{M})}{2}\right)^2 - \det(\bar{M})}$$
$$= \frac{1}{2}\left(\bar{A} + \bar{B} \pm \sqrt{\bar{A}^2 - 2\bar{A}\bar{B} + \bar{B}^2 + 4\bar{C}^2}\right)$$

- For corner, second edge must be significant (corresponds to trace(M)/2 - $\sqrt{\ldots}$)
- Function for corner detection:

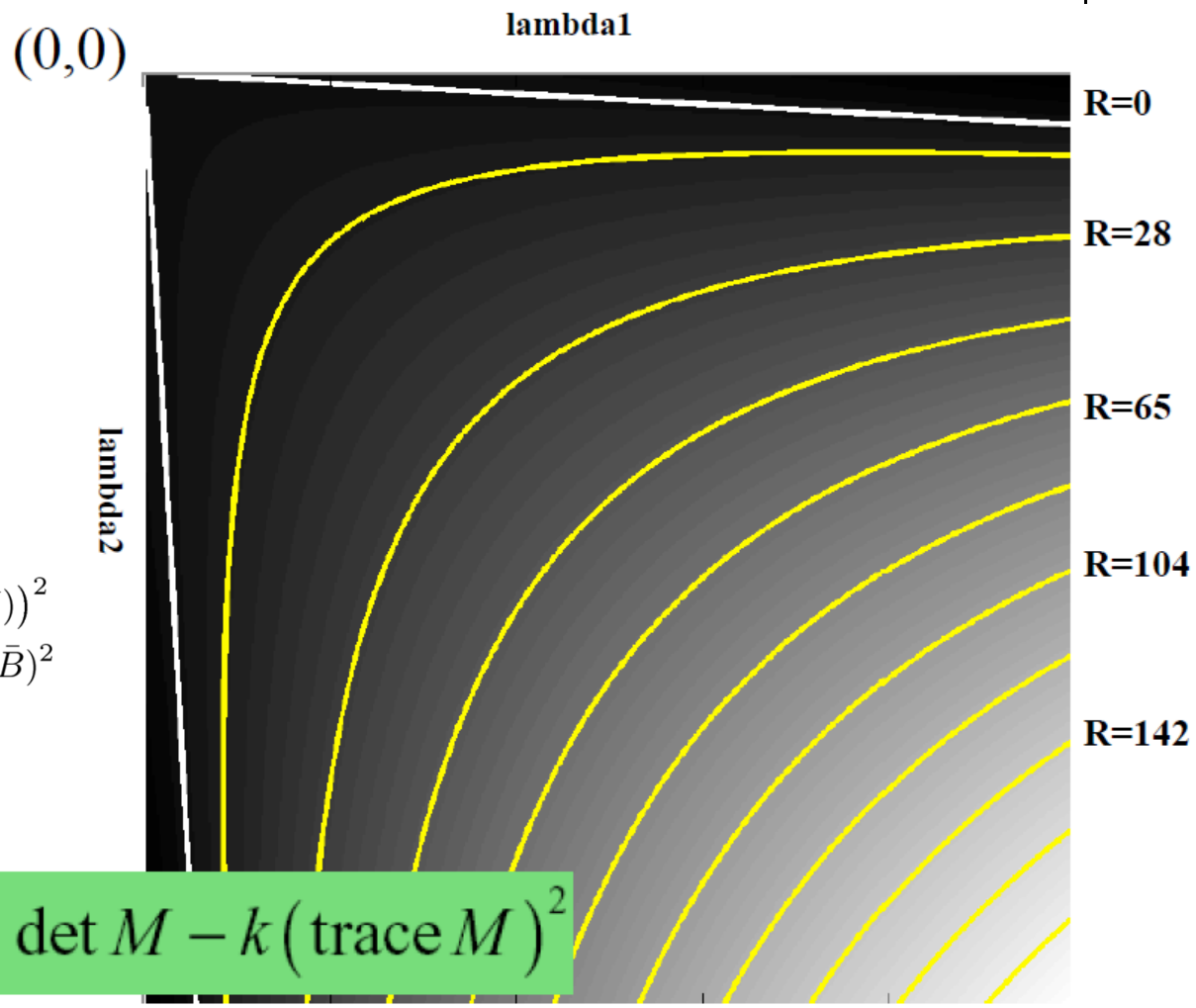
$$\bar{M}' = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

$$Q(u,v) = \det(\bar{M}) - \alpha \cdot \left(\text{trace}(\bar{M})\right)^2$$
$$= (\bar{A}\bar{B} - \bar{C}^2) - \alpha \cdot (\bar{A} + \bar{B})^2$$

$$\det M = \lambda_1 \lambda_2$$
$$\text{trace}\, M = \lambda_1 + \lambda_2$$

- In practice α assigned fixed value in range 0.04 – 0.06 (max 0.25)
- Larger values of α makes detector function less sensitive (fewer

Plot of Harris Corner Response Function



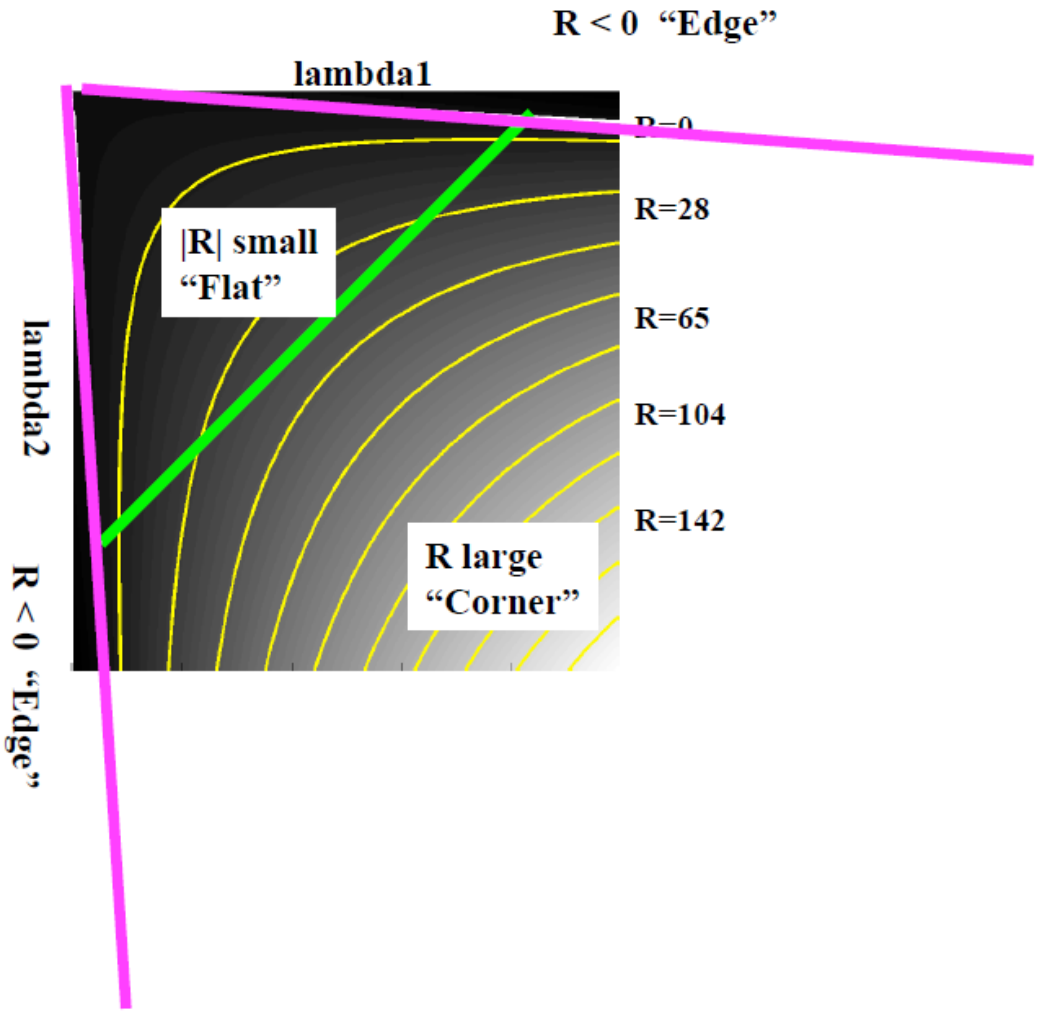
$$Q(u,v) = \det(\bar{M}) - \alpha \cdot \left(\text{trace}(\bar{M})\right)^2$$
$$= (\bar{A}\bar{B} - \bar{C}^2) - \alpha \cdot (\bar{A} + \bar{B})^2$$

Note: Same function

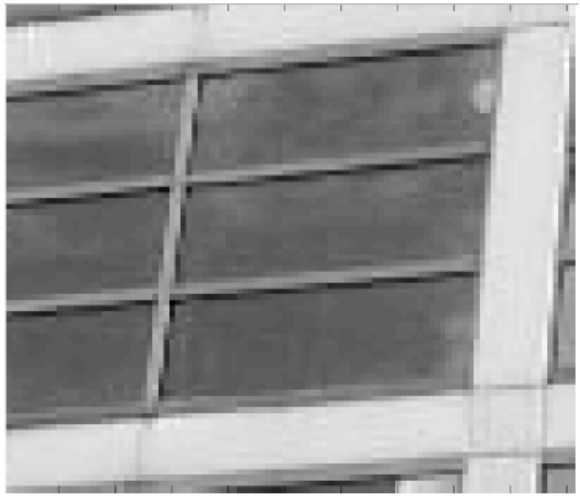
$$R = \det M - k\left(\text{trace}\, M\right)^2$$

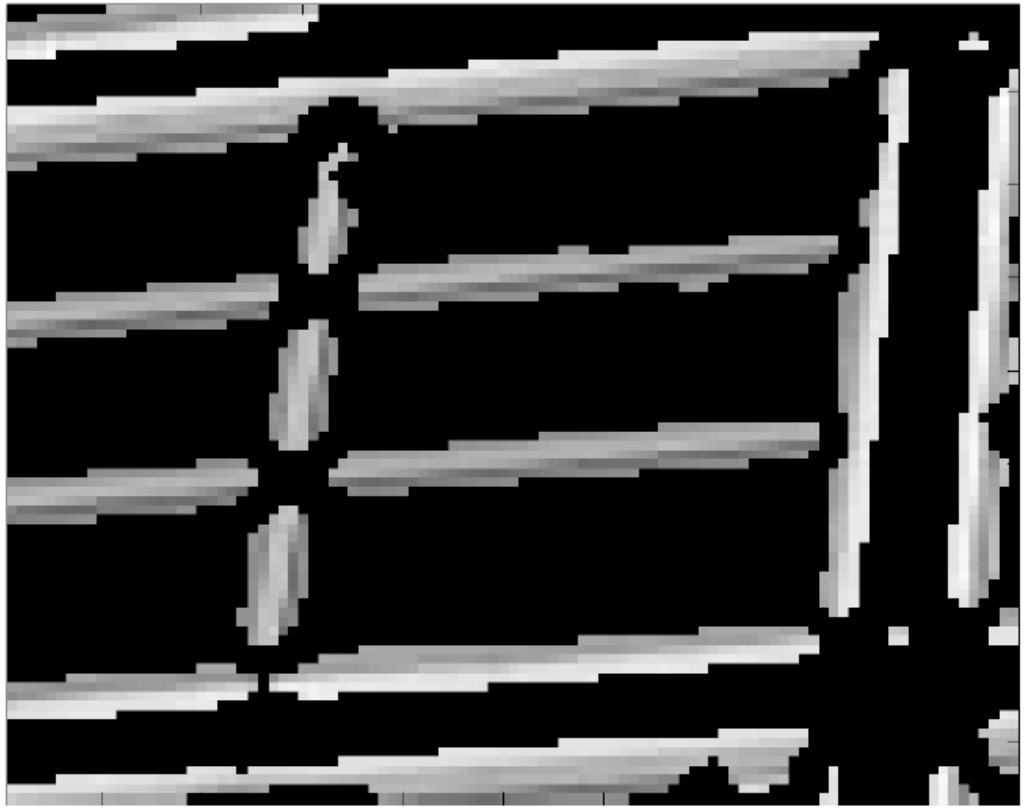
Plot of Harris Corner Response Function

- R depends only on eigenvalues of M
- R is large for a corner
- R is negative with large magnitude for an edge
- $|R|$ is small for a flat region



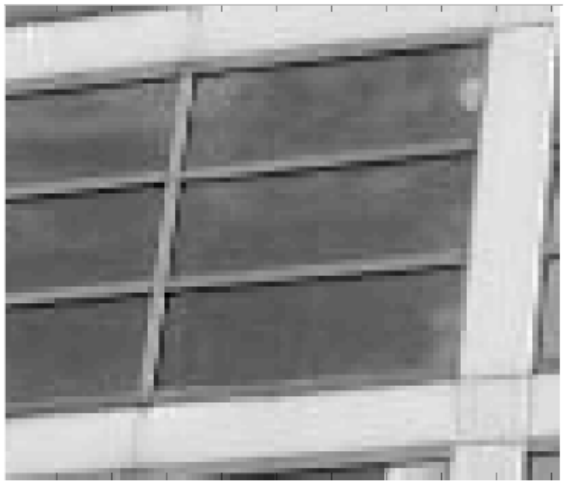
Harris Corner Response Example

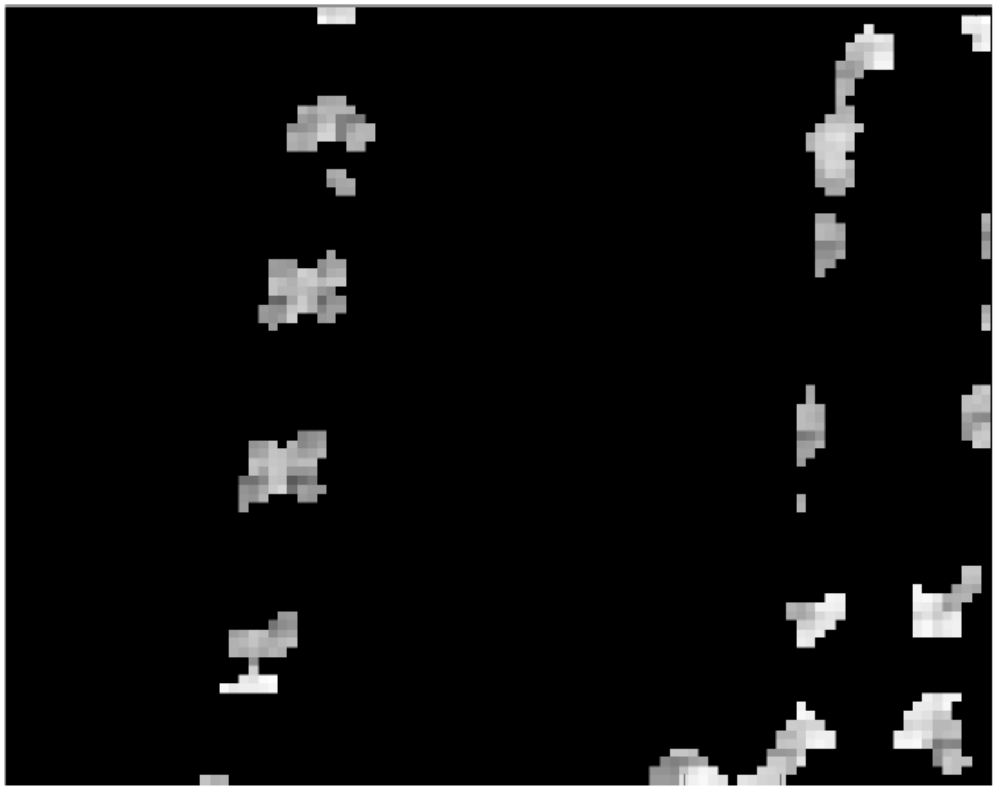




Threshold: $R < -10000$
(edges)

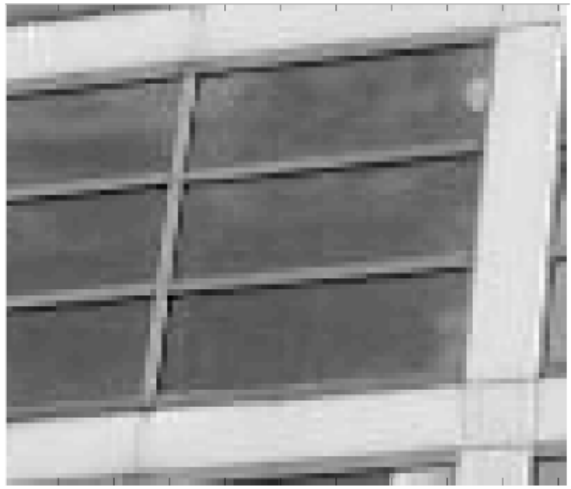
Harris Corner Response Example

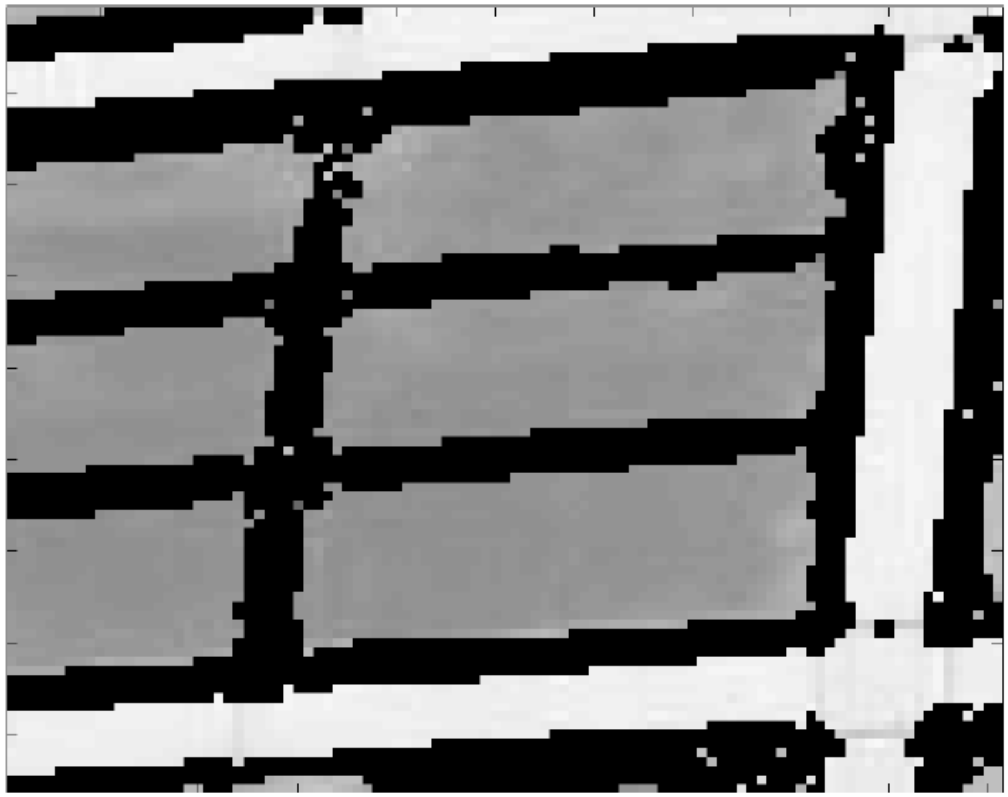




Threshold: > 10000
(corners)

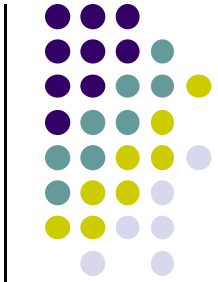
Harris Corner Response Example





Threshold: $-10000 < R < 10000$
(neither edges nor corners)

Harris Corner Detector



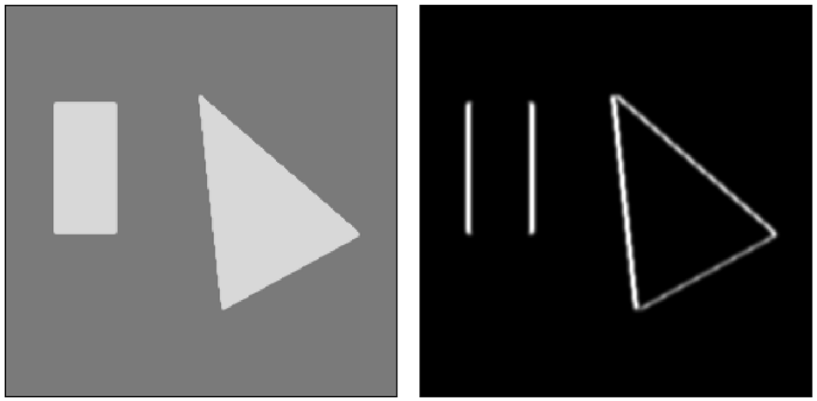
- An image location (*u,v*) is candidate for corner point when

$$Q(u, v) > t_H$$

- t_H is threshold selected based on image content, typically lies in range 10,000 – 1,000,000
- Detected corners inserted into set and sorted in descending order (i.e. $q_i \geq q_{i+1}$) based on their corner strength

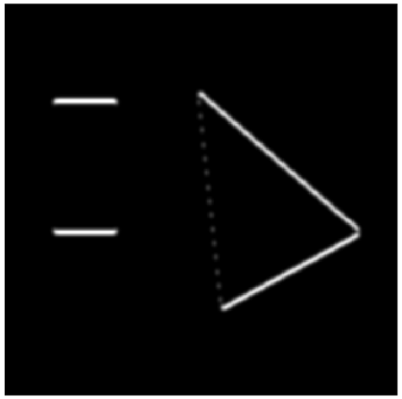
$$Corners = [\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots \boldsymbol{c}_N]$$

- Many false corners occur in neighborhood of real corner
- Traverse sorted list, delete false corners towards end of list

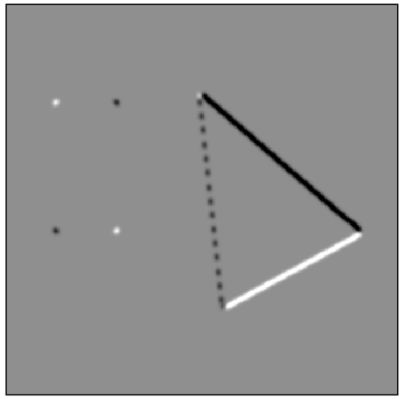


$I(u,v)$

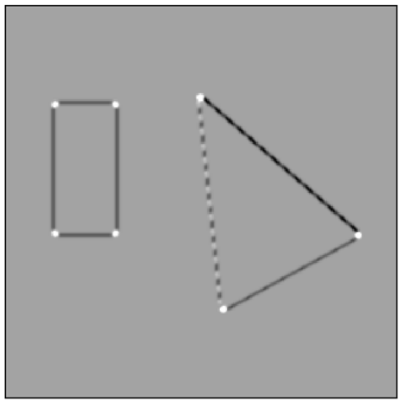
$A = I_x^2(u,v)$



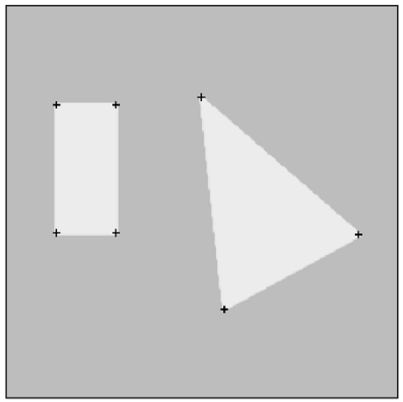
$B = I_y^2(u,v)$



$C = I_x I_y(u,v)$



$Q(u,v)$

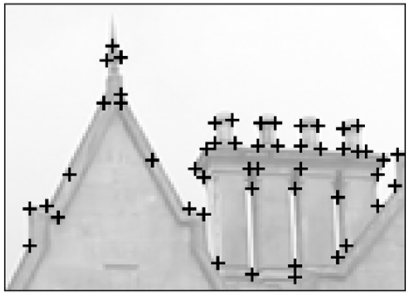


detected corners

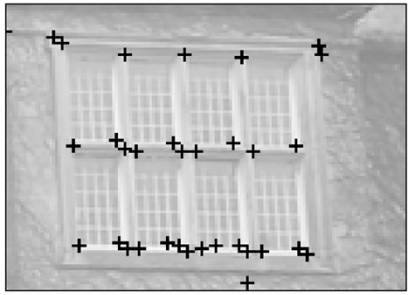
Harris Corner Detection Example (Synthetic Image)

(a)

Image with final corner points marked

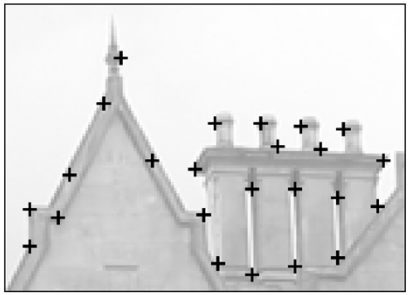


(b)

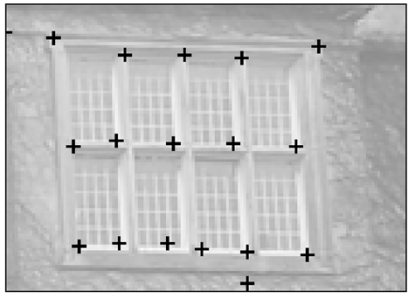


(c)

Before thresholding values



(d)



(e)

After thresholding

Harris Corner Detection Example (Real Image)

HarrisCorners(I)

Returns a list of the strongest corners found in the image I.

Step 1—compute the corner response function:

Prefilter (smooth) the original image: $I' \leftarrow I * H_p$

Compute the horizontal and vertical image derivatives:

$I_x \leftarrow I' * H_{dx}$

$I_y \leftarrow I' * H_{dy}$

Compute the local structure matrix $M(u,v) = \left(\begin{smallmatrix} A & C \\ C & B \end{smallmatrix}\right)$:

$A(u,v) \leftarrow I_x^2(u,v)$

$B(u,v) \leftarrow I_y^2(u,v)$

$C(u,v) \leftarrow I_x(u,v) \cdot I_y(u,v)$

Blur each component of the structure matrix: $\bar{M} = \left(\begin{smallmatrix} \bar{A} & \bar{C} \\ \bar{C} & \bar{B} \end{smallmatrix}\right)$:

$\bar{A} \leftarrow A * H_b$

$\bar{B} \leftarrow B * H_b$

$\bar{C} \leftarrow C * H_b$

Compute the corner response function:

$Q \leftarrow (\bar{A} \cdot \bar{B} - \bar{C}^2) - \alpha \cdot (\bar{A} + \bar{B})^2$

Step 2—collect corner points:

Create an empty list:

Corners $\leftarrow$ [ ]

for all image coordinates (u,v) **do**

if $Q(u,v) > t_H$ **and** IsLocalMax(Q,u,v) **then**

Create a new corner:

$\boldsymbol{c}_i \leftarrow \langle u_i, v_i, q_i \rangle = \langle u, v, Q(u,v) \rangle$

Add $\boldsymbol{c}_i$ to *Corners*

Sort *Corners* by q_i in *descending* order (strongest corners first)

GoodCorners $\leftarrow$ CleanUpNeighbors(*Corners*)

return *GoodCorners*.

IsLocalMax(Q,u,v) ▷ determine if $Q(u,v)$ is a local maximum

Let $q_c \leftarrow Q(u,v)$ (center pixel)

Let $\mathcal{N} \leftarrow$ *Neighbors*(Q,u,v) ▷ values of all neighboring pixels

if $q_c \geq q_i$ for all $q_i \in \mathcal{N}$ **then**

return *true*

else

return *false*.

CleanUpNeighbors(*Corners*) ▷ *Corners* is sorted by descending q

Create an empty list:

GoodCorners $\leftarrow$ [ ]

while *Corners* is not empty **do**

$\boldsymbol{c}_i \leftarrow$ RemoveFirst(*Corners*)

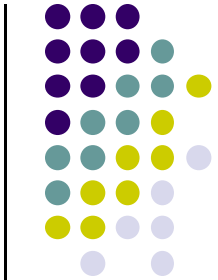
Add $\boldsymbol{c}_i$ to *GoodCorners*

for all $\boldsymbol{c}_j$ in *Corners* **do**

if $Dist(\boldsymbol{c}_i, \boldsymbol{c}_j) < d_{\min}$ **then**

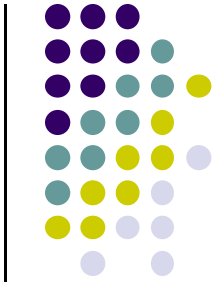
Delete $\boldsymbol{c}_j$ from *Corners*

return *GoodCorners*.



Harris Corner Detection Algorithm

Harris Corner Detection Algorithm & Parameters



```
1:  HarrisCorners(I)
      Returns a list of the strongest corners found in the image I.
2:    Step 1—compute the corner response function:
3:      Prefilter (smooth) the original image: I' ← I * H_p
4:      Compute the horizontal and vertical image derivatives:
          I_x ← I' * H_dx
          I_y ← I' * H_dy
5:      Compute the local structure matrix M(u,v) = (A C; C B):
          A(u,v) ← I_x^2(u,v)
          B(u,v) ← I_y^2(u,v)
          C(u,v) ← I_x(u,v) · I_y(u,v)
6:      Blur each component of the structure matrix: M̄ = (Ā C̄; C̄ B̄):
          Ā ← A * H_b
          B̄ ← B * H_b
          C̄ ← C * H_b
7:      Compute the corner response function:
          Q ← (Ā · B̄ − C̄^2) − α · (Ā + B̄)^2
8:    Step 2—collect corner points:
9:      Create an empty list:
          Corners ← [ ]
10:     for all image coordinates (u,v) do
11:       if Q(u,v) > t_H and IsLocalMax(Q,u,v) then
12:         Create a new corner:
              c_i ← ⟨u_i, v_i, q_i⟩ = ⟨u, v, Q(u,v)⟩
13:         Add c_i to Corners
14:     Sort Corners by q_i in descending order (strongest corners first)
15:     GoodCorners ← CleanUpNeighbors(Corners)
16:   return GoodCorners.

17: IsLocalMax(Q,u,v)        ▷ determine if Q(u,v) is a local maximum
18:   Let q_c ← Q(u,v) (center pixel)
19:   Let 𝒩 ← Neighbors(Q,u,v)        ▷ values of all neighboring pixels
20:   if q_c ≥ q_i for all q_i ∈ 𝒩 then
21:     return true
22:   else
23:     return false.

24: CleanUpNeighbors(Corners)    ▷ Corners is sorted by descending q
25:   Create an empty list:
        GoodCorners ← [ ]
26:   while Corners is not empty do
27:     c_i ← RemoveFirst(Corners)
28:     Add c_i to GoodCorners
29:     for all c_j in Corners do
30:       if Dist(c_i, c_j) < d_min then
31:         Delete c_j from Corners
32:   return GoodCorners.
```

Prefilter (line 3): Smoothing with a small xy-separable filter $H_p = H_{px} * H_{py}$, where

$$H_{px} = \frac{1}{9}\begin{bmatrix}2 & \mathbf{5} & 2\end{bmatrix} \quad \text{and} \quad H_{py} = H_{px}^T = \frac{1}{9}\begin{bmatrix}2\\ \mathbf{5}\\ 2\end{bmatrix}.$$

Gradient filter (line 4): Computing the first partial derivative in the x and y directions with

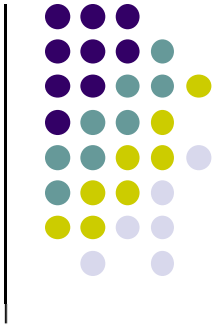
$$H_{dx} = \begin{bmatrix}-0.453014 & \mathbf{0} & 0.453014\end{bmatrix} \quad \text{and} \quad H_{dy} = H_{dx}^T = \begin{bmatrix}-0.453014\\ \mathbf{0}\\ 0.453014\end{bmatrix}.$$

Blurfilter (line 6): Smoothing the individual components of the structure matrix M with separable Gaussian filters $H_b = H_{bx} * H_{by}$ with

$$H_{bx} = \frac{1}{64}\begin{bmatrix}1 & 6 & 15 & \mathbf{20} & 15 & 6 & 1\end{bmatrix}, \qquad H_{by} = H_{bx}^T = \frac{1}{64}\begin{bmatrix}1\\ 6\\ 15\\ \mathbf{20}\\ 15\\ 6\\ 1\end{bmatrix}.$$

Steering parameter (line 7): $\alpha = 0.04$ to 0.06 (default 0.05)
Response threshold (line 13): $t_H = 10{,}000$ to $1{,}000{,}000$ (default $25{,}000$)
Neighborhood radius (line 31): $d_{\min} = 10$ pixels

Step 1: Computing Harris Corner Response Function



Prefilter (line 3): Smoothing with a small xy-separable filter $H_p = H_{px} * H_{py}$, where

$$H_{px} = \frac{1}{9}\begin{bmatrix}2 & \mathbf{5} & 2\end{bmatrix} \quad \text{and} \quad H_{py} = H_{px}^T = \frac{1}{9}\begin{bmatrix}2\\ \mathbf{5}\\ 2\end{bmatrix}.$$

Gradient filter (line 4): Computing the first partial derivative in the x and y directions with

$$H_{dx} = \begin{bmatrix}-0.453014 & \mathbf{0} & 0.453014\end{bmatrix} \quad \text{and} \quad H_{dy} = H_{dx}^T = \begin{bmatrix}-0.453014\\ \mathbf{0}\\ 0.453014\end{bmatrix}.$$

Blurfilter (line 6): Smoothing the individual components of the structure matrix M with separable Gaussian filters $H_b = H_{bx} * H_{by}$ with

$$H_{bx} = \frac{1}{64}\begin{bmatrix}1 & 6 & 15 & \mathbf{20} & 15 & 6 & 1\end{bmatrix}, \qquad H_{by} = H_{bx}^T = \frac{1}{64}\begin{bmatrix}1\\ 6\\ 15\\ \mathbf{20}\\ 15\\ 6\\ 1\end{bmatrix}.$$

Steering parameter (line 7): $\alpha = 0.04$ to 0.06 (default 0.05)
Response threshold (line 13): $t_H = 10{,}000$ to $1{,}000{,}000$ (default $25{,}000$)
Neighborhood radius (line 31): $d_{\min} = 10$ pixels

Declare filter values

```
1 float[] pfilt = {0.223755f,0.552490f,0.223755f}; // Hp
2 float[] dfilt = {0.453014f,0.0f,-0.453014f}; // Hdx, Hdy
3 float[] bfilt = {0.01563f,0.09375f,0.234375f,0.3125f,
4                  0.234375f,0.09375f,0.01563f}; // Hb
```

Create copies of Image

```
5 FloatProcessor Ix = (FloatProcessor) ip.convertToFloat();
6 FloatProcessor Iy = (FloatProcessor) ip.convertToFloat();
```

1: HarrisCorners(I)

Returns a list of the strongest corners found in the image I.

2: Step 1—compute the corner response function:

3: Prefilter (smooth) the original image: $I' \leftarrow I * H_p$

4: Compute the horizontal and vertical image derivatives:

$I_x \leftarrow I' * H_{dx}$

$I_y \leftarrow I' * H_{dy}$

5: Compute the local structure matrix: $M(u,v) = \left(\begin{smallmatrix} A & C \\ C & B \end{smallmatrix}\right)$:

$A(u,v) \leftarrow I_x^2(u,v)$

$B(u,v) \leftarrow I_y^2(u,v)$

$C(u,v) \leftarrow I_x(u,v) \cdot I_y(u,v)$

6: Blur each component of the structure matrix: $\bar{M} = \left(\begin{smallmatrix} \bar{A} & \bar{C} \\ \bar{C} & \bar{B} \end{smallmatrix}\right)$:

$\bar{A} \leftarrow A * H_b$

$\bar{B} \leftarrow B * H_b$

$\bar{C} \leftarrow C * H_b$

7: Compute the corner response function:

$Q \leftarrow (\bar{A} \cdot \bar{B} - \bar{C}^2) - \alpha \cdot (\bar{A} + \bar{B})^2$

8: Step 2—collect corner points:

9: Create an empty list:

Corners $\leftarrow [\,]$

10: **for** all image coordinates (u,v) **do**

11: **if** $Q(u,v) > t_H$ **and** IsLocalMax(Q,u,v) **then**

12: Create a new corner:

$\boldsymbol{c}_i \leftarrow \langle u_i, v_i, q_i \rangle = \langle u, v, Q(u,v) \rangle$

13: Add $\boldsymbol{c}_i$ to *Corners*

14: **Sort** *Corners* by q_i in *descending* order (strongest corners first)

15: *GoodCorners* $\leftarrow$ CleanUpNeighbors(*Corners*)

16: **return** *GoodCorners*.

17: IsLocalMax(Q,u,v) ▷ determine if $Q(u,v)$ is a local maximum

18: Let $q_c \leftarrow Q(u,v)$ (center pixel)

19: Let $\mathcal{N} \leftarrow$ *Neighbors*(Q,u,v) ▷ values of all neighboring pixels

20: **if** $q_c \geq q_i$ for all $q_i \in \mathcal{N}$ **then**

21: **return** *true*

22: **else**

23: **return** *false*.

24: CleanUpNeighbors(*Corners*) ▷ *Corners* is sorted by descending q

25: Create an empty list:

GoodCorners $\leftarrow [\,]$

26: **while** *Corners* is not empty **do**

27: $\boldsymbol{c}_i \leftarrow$ RemoveFirst(*Corners*)

28: Add $\boldsymbol{c}_i$ to *GoodCorners*

29: **for** all $\boldsymbol{c}_j$ in *Corners* **do**

30: **if** *Dist*$(\boldsymbol{c}_i, \boldsymbol{c}_j) < d_{\min}$ **then**

31: Delete $\boldsymbol{c}_j$ from *Corners*

32: **return** *GoodCorners*.

Apply pre-smoothing + gradient computation in one combined step

```
 7 Ix = convolve1h(convolve1h(Ix,pfilt),dfilt);
 8 Iy = convolve1v(convolve1v(Iy,pfilt),dfilt);
```

Compute components A, B, C and smooth

```
 9 A = sqr ((FloatProcessor) Ix.duplicate());
10 B = sqr ((FloatProcessor) Iy.duplicate());
11 C = mult((FloatProcessor) Ix.duplicate(),Iy);
12
13 A = convolve2(A,bfilt);    // convolve with Hb
14 B = convolve2(B,bfilt);
15 C = convolve2(C,bfilt);
```

Step 1: Computing Harris Corner Response Function

1: HarrisCorners(I)

Returns a list of the strongest corners found in the image I.

2: Step 1—compute the corner response function:

3: Prefilter (smooth) the original image: $I' \leftarrow I * H_p$

4: Compute the horizontal and vertical image derivatives:

$I_x \leftarrow I' * H_{dx}$

$I_y \leftarrow I' * H_{dy}$

5: Compute the local structure matrix $M(u,v) = \left(\begin{smallmatrix} A & C \\ C & B \end{smallmatrix}\right)$:

$A(u,v) \leftarrow I_x^2(u,v)$

$B(u,v) \leftarrow I_y^2(u,v)$

$C(u,v) \leftarrow I_x(u,v) \cdot I_y(u,v)$

6: Blur each component of the structure matrix: $\bar{M} = \left(\begin{smallmatrix} \bar{A} & \bar{C} \\ \bar{C} & \bar{B} \end{smallmatrix}\right)$:

$\bar{A} \leftarrow A * H_b$

$\bar{B} \leftarrow B * H_b$

$\bar{C} \leftarrow C * H_b$

7: Compute the corner response function:

$Q \leftarrow (\bar{A} \cdot \bar{B} - \bar{C}^2) - \alpha \cdot (\bar{A} + \bar{B})^2$

8: Step 2—collect corner points:

9: Create an empty list:

Corners $\leftarrow [\,]$

10: **for** all image coordinates (u,v) **do**

11: **if** $Q(u,v) > t_H$ **and** IsLocalMax(Q,u,v) **then**

12: Create a new corner:

$\boldsymbol{c}_i \leftarrow \langle u_i, v_i, q_i \rangle = \langle u, v, Q(u,v) \rangle$

13: Add $\boldsymbol{c}_i$ to *Corners*

14: **Sort** *Corners* by q_i in *descending* order (strongest corners first)

15: *GoodCorners* $\leftarrow$ CleanUpNeighbors(*Corners*)

16: **return** *GoodCorners*.

17: IsLocalMax(Q,u,v) ▷ determine if $Q(u,v)$ is a local maximum

18: Let $q_c \leftarrow Q(u,v)$ (center pixel)

19: Let $\mathcal{N} \leftarrow$ *Neighbors*(Q,u,v) ▷ values of all neighboring pixels

20: **if** $q_c \geq q_i$ for all $q_i \in \mathcal{N}$ **then**

21: **return** *true*

22: **else**

23: **return** *false*.

24: CleanUpNeighbors(*Corners*) ▷ *Corners* is sorted by descending q

25: Create an empty list:

GoodCorners $\leftarrow [\,]$

26: **while** *Corners* is not empty **do**

27: $\boldsymbol{c}_i \leftarrow$ RemoveFirst(*Corners*)

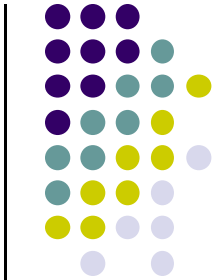
28: Add $\boldsymbol{c}_i$ to *GoodCorners*

29: **for** all $\boldsymbol{c}_j$ in *Corners* **do**

30: **if** *Dist*$(\boldsymbol{c}_i, \boldsymbol{c}_j) < d_{\min}$ **then**

31: Delete $\boldsymbol{c}_j$ from *Corners*

32: **return** *GoodCorners*.

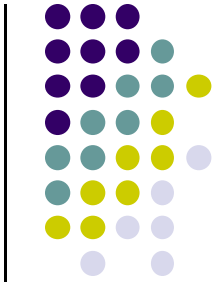


Compute Corner Response Function

```
16 void makeCrf() { // defined in class HarrisCornerDetector
17   int w = ipOrig.getWidth();
18   int h = ipOrig.getHeight();
19   Q = new FloatProcessor(w,h);
20   float[] Apix = (float[]) A.getPixels();
21   float[] Bpix = (float[]) B.getPixels();
22   float[] Cpix = (float[]) C.getPixels();
23   float[] Qpix = (float[]) Q.getPixels();
24   for (int v=0; v<h; v++) {
25     for (int u=0; u<w; u++) {
26       int i = v*w+u;
27       float a = Apix[i], b = Bpix[i], c = Cpix[i];
28       float det = a*b-c*c;          // det(M̄)
29       float trace = a+b;            // trace(M̄)
30       Qpix[i] = det - alpha * (trace * trace);
31     }
32   }
33 }
```

Step 1: Computing Harris Corner Response Function

```
1:  HarrisCorners(I)
      Returns a list of the strongest corners found in the image I.
2:    Step 1—compute the corner response function:
3:      Prefilter (smooth) the original image: I' ← I * H_p
4:      Compute the horizontal and vertical image derivatives:
          I_x ← I' * H_dx
          I_y ← I' * H_dy
5:      Compute the local structure matrix M(u,v) = ( A C ; C B ):
          A(u,v) ← I_x^2(u,v)
          B(u,v) ← I_y^2(u,v)
          C(u,v) ← I_x(u,v) · I_y(u,v)
6:      Blur each component of the structure matrix: M̄ = ( Ā C̄ ; C̄ B̄ ):
          Ā ← A * H_b
          B̄ ← B * H_b
          C̄ ← C * H_b
7:      Compute the corner response function:
          Q ← (Ā · B̄ − C̄^2) − α · (Ā + B̄)^2
8:    Step 2—collect corner points:
9:      Create an empty list:
          Corners ← [ ]
10:     for all image coordinates (u,v) do
11:       if Q(u,v) > t_H and IsLocalMax(Q,u,v) then
12:         Create a new corner:
              c_i ← ⟨u_i, v_i, q_i⟩ = ⟨u, v, Q(u,v)⟩
13:         Add c_i to Corners
14:     Sort Corners by q_i in descending order (strongest corners first)
15:     GoodCorners ← CleanUpNeighbors(Corners)
16:   return GoodCorners.

17: IsLocalMax(Q,u,v)        ▷ determine if Q(u,v) is a local maximum
18:   Let q_c ← Q(u,v) (center pixel)
19:   Let 𝒩 ← Neighbors(Q,u,v)        ▷ values of all neighboring pixels
20:   if q_c ≥ q_i for all q_i ∈ 𝒩 then
21:     return true
22:   else
23:     return false.

24: CleanUpNeighbors(Corners)    ▷ Corners is sorted by descending q
25:   Create an empty list:
        GoodCorners ← [ ]
26:   while Corners is not empty do
27:     c_i ← RemoveFirst(Corners)
28:     Add c_i to GoodCorners
29:     for all c_j in Corners do
30:       if Dist(c_i, c_j) < d_min then
31:         Delete c_j from Corners
32:   return GoodCorners.
```



Declare Corners

```
50 public class Corner implements Comparable {
51   int u;    // x position
52   int v;    // y position
53   float q; // corner strength
54
55   Corner (int u, int v, float q) { //constructor method
56     this.u = u;
57     this.v = v;
58     this.q = q;
59   }
60 }
```

Step 2: Selecting "Good" Corner Points

1: HarrisCorners(I)

Returns a list of the strongest corners found in the image I.

2: Step 1—compute the corner response function:

3: Prefilter (smooth) the original image: $I' \leftarrow I * H_p$

4: Compute the horizontal and vertical image derivatives:

$I_x \leftarrow I' * H_{dx}$

$I_y \leftarrow I' * H_{dy}$

5: Compute the local structure matrix $M(u,v) = \left(\begin{smallmatrix} A & C \\ C & B \end{smallmatrix}\right)$:

$A(u,v) \leftarrow I_x^2(u,v)$

$B(u,v) \leftarrow I_y^2(u,v)$

$C(u,v) \leftarrow I_x(u,v) \cdot I_y(u,v)$

6: Blur each component of the structure matrix: $\bar{M} = \left(\begin{smallmatrix} \bar{A} & \bar{C} \\ \bar{C} & \bar{B} \end{smallmatrix}\right)$:

$\bar{A} \leftarrow A * H_b$

$\bar{B} \leftarrow B * H_b$

$\bar{C} \leftarrow C * H_b$

7: Compute the corner response function:

$Q \leftarrow (\bar{A} \cdot \bar{B} - \bar{C}^2) - \alpha \cdot (\bar{A} + \bar{B})^2$

8: Step 2—collect corner points:

9: Create an empty list:

Corners $\leftarrow [\,]$

10: **for** all image coordinates (u,v) **do**

11: **if** $Q(u,v) > t_H$ **and** IsLocalMax(Q,u,v) **then**

12: Create a new corner:

$\boldsymbol{c}_i \leftarrow \langle u_i, v_i, q_i \rangle = \langle u, v, Q(u,v) \rangle$

13: Add $\boldsymbol{c}_i$ to *Corners*

14: **Sort** *Corners* by q_i in *descending* order (strongest corners first)

15: *GoodCorners* $\leftarrow$ CleanUpNeighbors(*Corners*)

16: **return** *GoodCorners*.

17: IsLocalMax(Q,u,v) ▷ determine if $Q(u,v)$ is a local maximum

18: Let $q_c \leftarrow Q(u,v)$ (center pixel)

19: Let $\mathcal{N} \leftarrow$ *Neighbors*(Q,u,v) ▷ values of all neighboring pixels

20: **if** $q_c \geq q_i$ for all $q_i \in \mathcal{N}$ **then**

21: **return** *true*

22: **else**

23: **return** *false*.

24: CleanUpNeighbors(*Corners*) ▷ *Corners* is sorted by descending q

25: Create an empty list:

GoodCorners $\leftarrow [\,]$

26: **while** *Corners* is not empty **do**

27: $\boldsymbol{c}_i \leftarrow$ RemoveFirst(*Corners*)

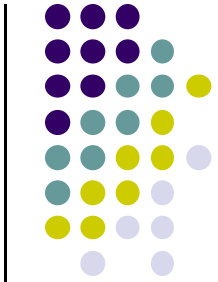
28: Add $\boldsymbol{c}_i$ to *GoodCorners*

29: **for** all $\boldsymbol{c}_j$ in *Corners* **do**

30: **if** *Dist*$(\boldsymbol{c}_i, \boldsymbol{c}_j) < d_{\min}$ **then**

31: Delete $\boldsymbol{c}_j$ from *Corners*

32: **return** *GoodCorners*.



Test image coordinates for Corners & sort

```
61 List<Corner> collectCorners(FloatProcessor Q, int border) {
62   List<Corner> cornerList = new Vector<Corner>(1000);
63   int w = Q.getWidth();
64   int h = Q.getHeight();
65   float[] Qpix = (float[]) Q.getPixels();
66   // traverse the Q-image and check for corners:
67   for (int v = border; v < h-border; v++){
68     for (int u = border; u < w-border; u++) {
69       float q = Qpix[v*w+u];
70       if (q > threshold && isLocalMax(crf,u,v)) {
71         Corner c = new Corner(u,v,q);
72         cornerList.add(c);
73       }
74     }
75   }
76   Collections.sort(cornerList);
77   return cornerList;
78 }
```

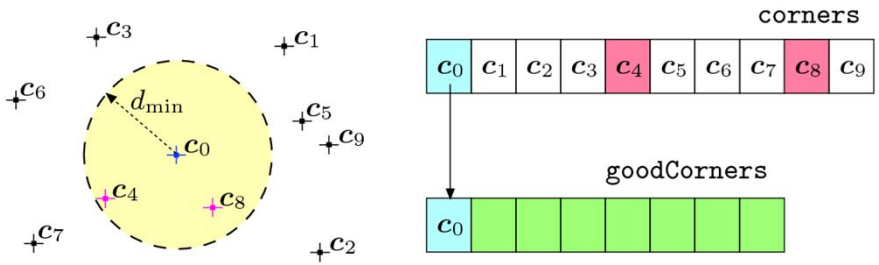
Step 2: Selecting "Good" Corner Points

```
1:  HarrisCorners(I)
      Returns a list of the strongest corners found in the image I.
2:    Step 1—compute the corner response function:
3:      Prefilter (smooth) the original image: I' ← I * H_p
4:      Compute the horizontal and vertical image derivatives:
          I_x ← I' * H_dx
          I_y ← I' * H_dy
5:      Compute the local structure matrix M(u,v) = (A C; C B):
          A(u,v) ← I_x^2(u,v)
          B(u,v) ← I_y^2(u,v)
          C(u,v) ← I_x(u,v) · I_y(u,v)
6:      Blur each component of the structure matrix: M̄ = (Ā C̄; C̄ B̄):
          Ā ← A * H_b
          B̄ ← B * H_b
          C̄ ← C * H_b
7:      Compute the corner response function:
          Q ← (Ā · B̄ − C̄^2) − α · (Ā + B̄)^2
8:    Step 2—collect corner points:
9:      Create an empty list:
          Corners ← [ ]
10:     for all image coordinates (u,v) do
11:       if Q(u,v) > t_H and IsLocalMax(Q,u,v) then
12:         Create a new corner:
              c_i ← ⟨u_i, v_i, q_i⟩ = ⟨u, v, Q(u,v)⟩
13:         Add c_i to Corners
14:     Sort Corners by q_i in descending order (strongest corners first)
15:     GoodCorners ← CleanUpNeighbors(Corners)
16:   return GoodCorners.

17: IsLocalMax(Q,u,v)        ▷ determine if Q(u,v) is a local maximum
18:   Let q_c ← Q(u,v) (center pixel)
19:   Let 𝒩 ← Neighbors(Q,u,v)        ▷ values of all neighboring pixels
20:   if q_c ≥ q_i for all q_i ∈ 𝒩 then
21:     return true
22:   else
23:     return false.

24: CleanUpNeighbors(Corners)    ▷ Corners is sorted by descending q
25:   Create an empty list:
        GoodCorners ← [ ]
26:   while Corners is not empty do
27:     c_i ← RemoveFirst(Corners)
28:     Add c_i to GoodCorners
29:     for all c_j in Corners do
30:       if Dist(c_i, c_j) < d_min then
31:         Delete c_j from Corners
32:   return GoodCorners.
```

Delete weak corners

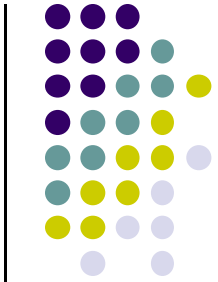
1. c_4 is added to new list goodCorners

2. c_4 and c_8 (weak corners within Circle of radius d_{min}) are removed



Step 2: Selecting "Good" Corner Points

Step 2: Selecting "Good" Corner Points



HarrisCorners(I)

Returns a list of the strongest corners found in the image I.

Step 1—compute the corner response function:

Prefilter (smooth) the original image: $I' \leftarrow I * H_p$

Compute the horizontal and vertical image derivatives:

$I_x \leftarrow I' * H_{dx}$

$I_y \leftarrow I' * H_{dy}$

Compute the local structure matrix $M(u,v) = \left(\begin{smallmatrix} A & C \\ C & B \end{smallmatrix}\right)$:

$A(u,v) \leftarrow I_x^2(u,v)$

$B(u,v) \leftarrow I_y^2(u,v)$

$C(u,v) \leftarrow I_x(u,v) \cdot I_y(u,v)$

Blur each component of the structure matrix: $\bar{M} = \left(\begin{smallmatrix} \bar{A} & \bar{C} \\ \bar{C} & \bar{B} \end{smallmatrix}\right)$:

$\bar{A} \leftarrow A * H_b$

$\bar{B} \leftarrow B * H_b$

$\bar{C} \leftarrow C * H_b$

Compute the corner response function:

$Q \leftarrow (\bar{A} \cdot \bar{B} - \bar{C}^2) - \alpha \cdot (\bar{A} + \bar{B})^2$

Step 2—collect corner points:

Create an empty list:

Corners $\leftarrow$ [ ]

for all image coordinates (u,v) **do**

if $Q(u,v) > t_H$ **and** IsLocalMax(Q,u,v) **then**

Create a new corner:

$\boldsymbol{c}_i \leftarrow \langle u_i, v_i, q_i \rangle = \langle u, v, Q(u,v) \rangle$

Add $\boldsymbol{c}_i$ to *Corners*

Sort *Corners* by q_i in *descending* order (strongest corners first)

GoodCorners $\leftarrow$ CleanUpNeighbors(*Corners*)

return *GoodCorners*.

IsLocalMax(Q,u,v) ▷ determine if $Q(u,v)$ is a local maximum

Let $q_c \leftarrow Q(u,v)$ (center pixel)

Let $\mathcal{N} \leftarrow$ *Neighbors*(Q,u,v) ▷ values of all neighboring pixels

if $q_c \geq q_i$ for all $q_i \in \mathcal{N}$ **then**

return *true*

else

return *false*.

CleanUpNeighbors(*Corners*) ▷ *Corners* is sorted by descending q

Create an empty list:

GoodCorners $\leftarrow$ [ ]

while *Corners* is not empty **do**

$\boldsymbol{c}_i \leftarrow$ RemoveFirst(*Corners*)

Add $\boldsymbol{c}_i$ to *GoodCorners*

for all $\boldsymbol{c}_j$ in *Corners* **do**

if *Dist*$(\boldsymbol{c}_i, \boldsymbol{c}_j) < d_{\min}$ **then**

Delete $\boldsymbol{c}_j$ from *Corners*

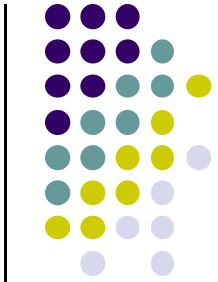
return *GoodCorners*.

Delete weak corners (code)

```
85 List<Corner> cleanupCorners(List<Corner> corners) {
86   // corners is assumed to be sorted by descending q
87   double dmin2 = dmin*dmin; // d²min (dmin is an object variable)
88   Corner[] cornerArray = new Corner[corners.size()];
89   cornerArray = corners.toArray(cornerArray);
90
91   List<Corner> goodCorners =
92                   new Vector<Corner>(corners.size());
93
94   for (int i = 0; i < cornerArray.length; i++){
95     if (cornerArray[i] != null) {
96       // select the next "good" corner c1
97       Corner c1 = cornerArray[i];
98       goodCorners.add(c1);
99       // remove all remaining corners too close to c1
100      for (int j = i+1; j < cornerArray.length; j++) {
101        if (cornerArray[j] != null) {
102          Corner c2 = cornerArray[j];
103          if (c1.dist2(c2) < dmin2) //compare squared distances
104            cornerArray[j] = null;  //remove corner c2
105        }
106      }
107    }
108  }
109  return goodCorners;
110 }
```

$$d^2(\boldsymbol{c}_1, \boldsymbol{c}_2) = (u_1 - u_2)^2 + (v_1 - v_2)^2$$

Harris Corner Detection: Run Method

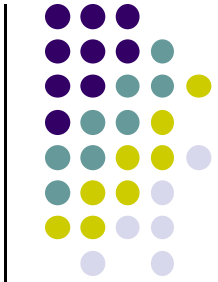


- Actual run method

```
139 public void run(ImageProcessor ip) {
140   HarrisCornerDetector hcd = new HarrisCornerDetector(ip);
141   hcd.findCorners();
142   ImageProcessor result = hcd.showCornerPoints(ip);
143   ImagePlus win = new ImagePlus("Corners",result);
144   win.show();
145 }
```

```
133 void findCorners(){              // defined in class Corner
134   makeDerivatives();
135   makeCrf();      // compute corner response function (CRF)
136   corners = collectCorners(border);
137   corners = cleanupCorners(corners);
138 }
```

References



- Wilhelm Burger and Mark J. Burge, Digital Image Processing, Springer, 2008
- Robert Collins, CSE 486 slides, Penn State University
- University of Utah, CS 4640: Image Processing Basics, Spring 2012
- Rutgers University, CS 334, Introduction to Imaging and Multimedia, Fall 2012
- Gonzales and Woods, Digital Image Processing (3rd edition), Prentice Hall