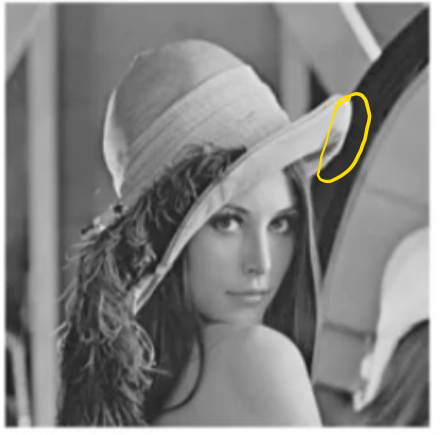
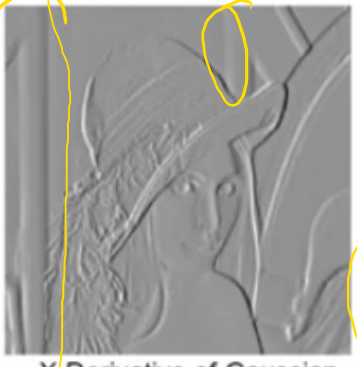
Compute of Gradients

Lina (image processing)

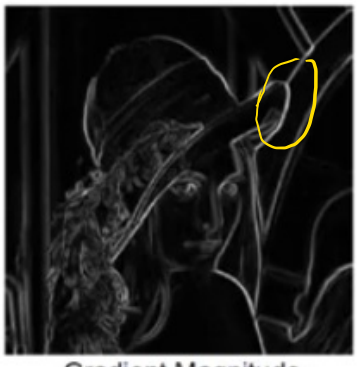


$$\|\nabla f\| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$



X-Derivative of Gaussian

Y-Derivative of Gaussian



Gradient Magnitude

You can compute the x derivative of the Gaussian

You can compute the y derivative of the Gaussian

Now take the gradient magnitude, which is root of x derivative square plus y derivative square and you will find that the gradient magnitude gives you a set of edges

Properties of an edge detector

When we take a closer look at one of these edges on the image that we just saw earlier. It is hard to find where is the edge really.

There seem to be so many pixels that are white here, that an edge could be anywhere on those pixels.

But when we say we want an edge, we expect certain properties to be met.

Properties of an edge detector

1 Good Detection

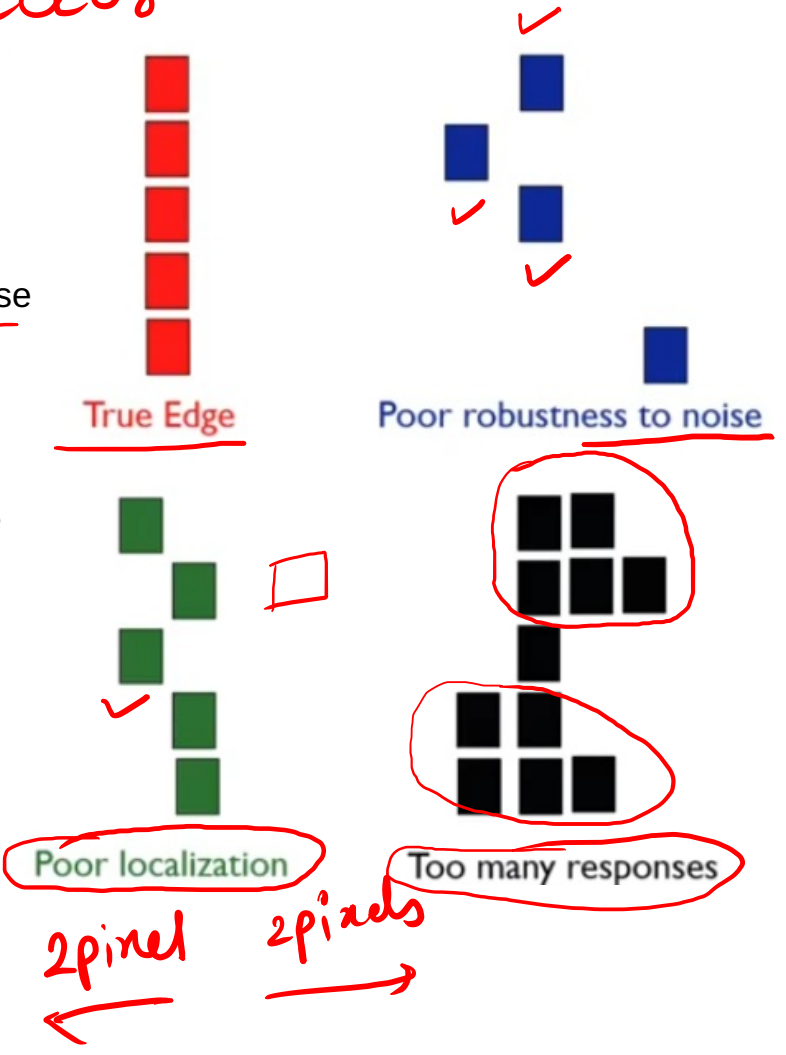
find all the real edges in the image and ignore noise or other artefacts. if there is an edge in the image, it should detect it. Noise points should not be detected as edges.

2 Good Localization

if the edge is in this location, you also want to detect the edge in the same location. If you are to detect edges in the vicinity in the neighborhood and not exactly where the edge is, that will be called poor localization, but what we want is good localization.

3 Single Response

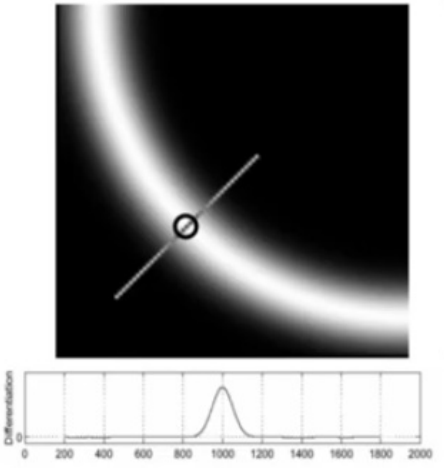
we only want a single response. We only want to return one point for a true edge point and not an entire region. If this was the true edge, this kind of a detection, which is spread out is not very useful and we want to be exactly at this location for a good edge detection method.

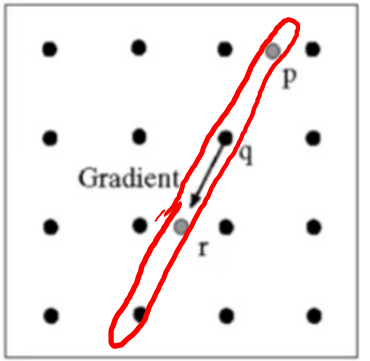


But so far, we only know how to use the gradient to detect the edges.

How do you solve problem such as good localization, single response, so on and so forth.

Non-Maxima Suppression





- Check if pixel is local maximum along gradient direction

A single response can be ensured in a particular region by doing Non-Maxima Suppression. Let us assume there was a circular kind of an edge in an image, that is how the edge looked.

Every white pixel would get detected as an edge, as we just saw, all of them could have high responses depending on what neighborhood you use to compute your gradient. So we may end up having many pixels corresponding to the edge, which you do not want. You want it to be isolated.

What do you do? You take a particular pixel, you compute the orientation of the gradient at that particular pixel as tan inverse of gradient along y direction by gradient along x direction, that gives you the orientation of the gradient at that particular pixel.

If you are at a pixel q and the orientation of the gradient is along a particular direction, then what you do is you go along the direction of the gradient and then try to see if this pixel is a maximum in that direction or not.

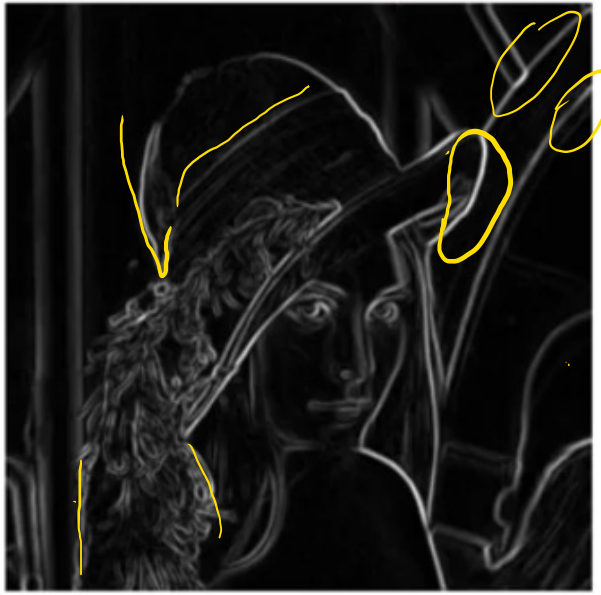
You want to retain only the pixel that has the highest gradient in value and make any other pixel that is not the maximum into zero. So you just check if the pixel is a local maximum along the gradient direction, if it is a maximum, retain its value, the edge magnitude value that you got, if it is not a maximum, make the edge magnitude zero and do not treat it as an edge.

upsampling

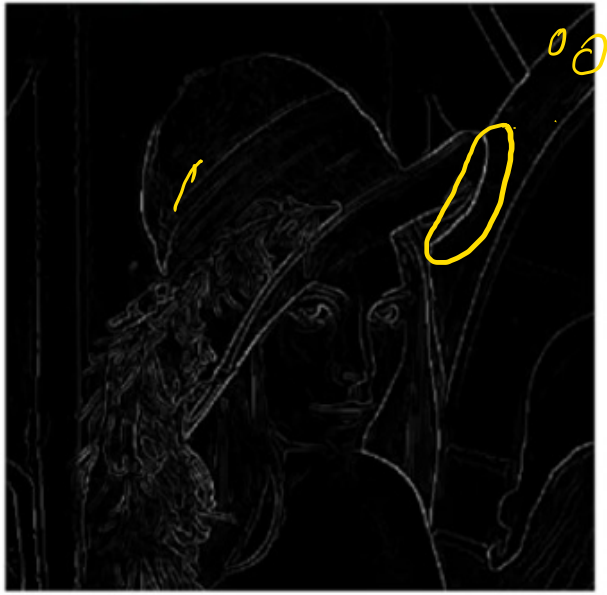
So in this process, one thing that you may have to remember, when you go along the direction of the gradient orientation, you may end up going to a location which is not defined on the image. It could be between two pixels. In those cases, you may have to interpolate, you can use simple linear interpolation or any simple interpolation method to be able to get what the value of the pixel of the gradient is at that location and you can then continue to do more maximum suppression.

Before and after Non-maxima Suppression

Lina Before



After Lina



fairly thinned
fairly localized

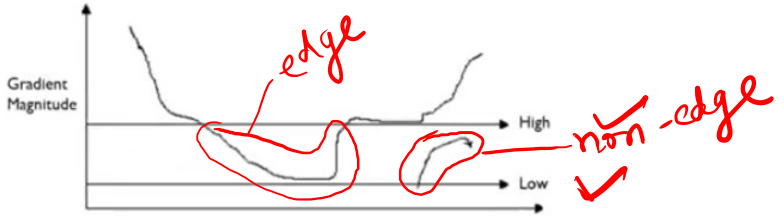
So here is an example of the same image after applying non-maximum suppression. And you can see now that the edges have fairly thinned out well and are fairly localized where the edge should be rather than have very thick edges.

One more problem that we find in the set of edges here is that, there are many discontinuities.

We would have ideally wanted the hat on her head to be one large edge, unfortunately, there seem to be multiple pieces.

In fact, there seem to be pieces even missing in between where we would have expected an edge.

Hysteresis Thresholding



Hysteresis Thresholding means that you would have to threshold your gradient magnitude

You would have two thresholds, a high threshold and a low threshold.

High gradient means it is an edge. If the gradient at a pixel is above the high threshold, it is definitely an edge pixel. Similarly, if the gradient at that pixel is less than a low value, it is definitely a non-edge pixel.

If you have the gradient at a particular pixel to be lying between these two thresholds. Then a particular pixel to be an edge pixel, if and only if, that pixel is connected to an edge pixel directly or via other pixels between low and high.

If you take one particular pixel there, you check if this pixel was connected directly or through a set of values to any edge pixel. If so, you would call that an edge pixel. On the other hand, if you take this pixel, if you now try to connect it, it does not connect to any edge pixel that is greater than high and hence, this entire set of pixels would get classified also as non-edge pixels.

Instead of relying on a threshold to be able to say whether you have an edge or not, you are now having two thresholds to ensure that even if you were on the boundary region, you could now use this approach to decide whether it should be an edge or not and that now gives more complete looking edges, where the edges are completely covering the object rather than those broken lines that we saw on the earlier.

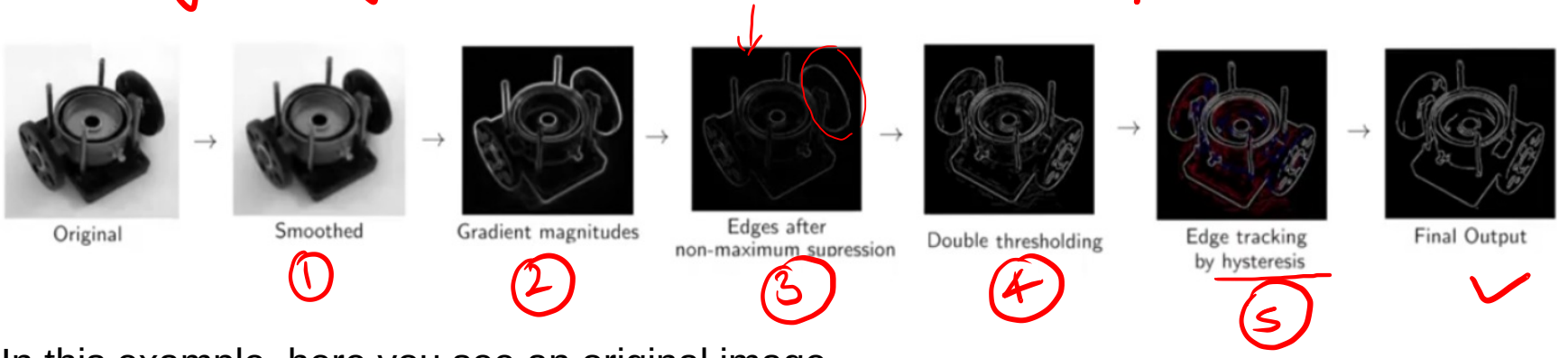
Canny Edge Detector

- Probably the most widely used edge detector in computer vision (J. Canny, 1986)
- Algorithm
  1. Filter image with derivative of Gaussian
  2. Find magnitude and orientation of gradient
  3. Non-maximum Suppression
  4. Linking and hysteresis Thresholding
     ↳ Use the high threshold to start edge curves and the low threshold to continue them.

So you define two thresholds, low and high.

You use the high threshold to start edge curves and you use the low threshold to ensure that those edge curves can be continued until a little bit lower to be able to get a sense of more connected edges in the image.

Canny Edge pipeline and Examples



In this example, here you see an original image,
1) it smoothen using a Gaussian filter
2) and here are the gradient magnitudes after applying derivative, and
3) then you have the edges after doing the non-maximum suppression, you can see that the edges have thinner.
4) Then you do the double thresholding, which is the high thresholding. So you keep only the edges, edge pixels that are greater than the high threshold and you remove anything lower than the low threshold.
5) Then you do the hysteresis to give the continuity of the edges in between, regions between high and low and here you have your final output after applying all of these processes.

other examples of canny edge detector

ex1

ex 2

dog

Here are a few more examples of applying Canny edges, and you can actually see that for these kinds of images, they do a fairly good job, in a fairly cohesive way across various artefacts in the image.

Effect of σ in Canny Edge Detector

original | Canny with $\sigma = 1$ | Canny with $\sigma = 2$

The choices of σ (Gaussian kernel size) depends on desired behaviour

(i) large σ detects large scale edges (ii) small σ detects fine edges

So one of the parameters that you have with the Canny Edge Detector is the size of the Gaussian kernel.

You could have various sigmas, you could have the Gaussian filter to be 3 cross 3, 5 cross 5, so on and so forth. So let us try to see what happens if you change the sigma in your Gaussian.

You will notice that a large sigma detects large scale edges and small sigma detects finer edges, it is very straightforward. So if you had a canny with sigma is equal to 1, you would have lots of small edges, whereas if you now increase sigma, then you would end up finding a canny filter, which has only large scale edges, not finer details, mainly because it increased the gradients of the Gaussian.

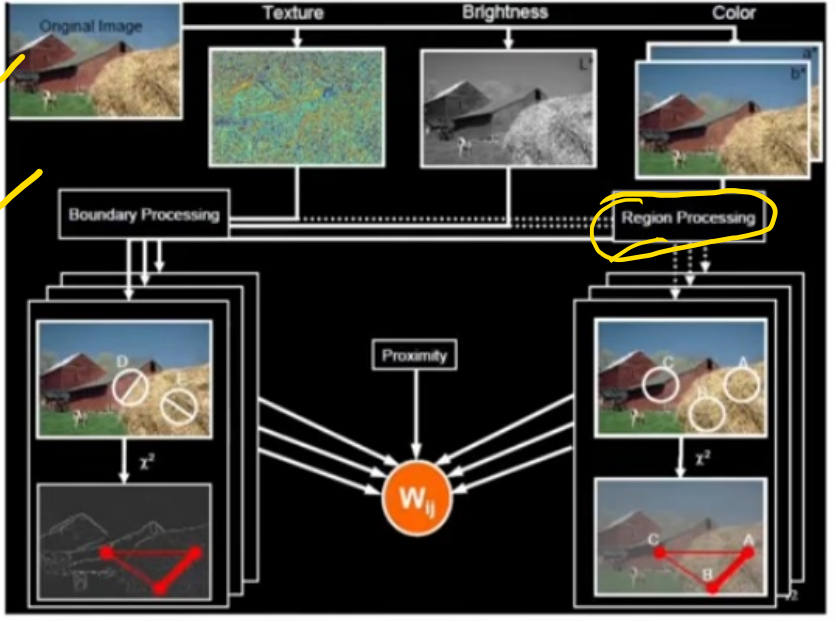
While Canny Edge Detector was the most popular edge detector for many decades.

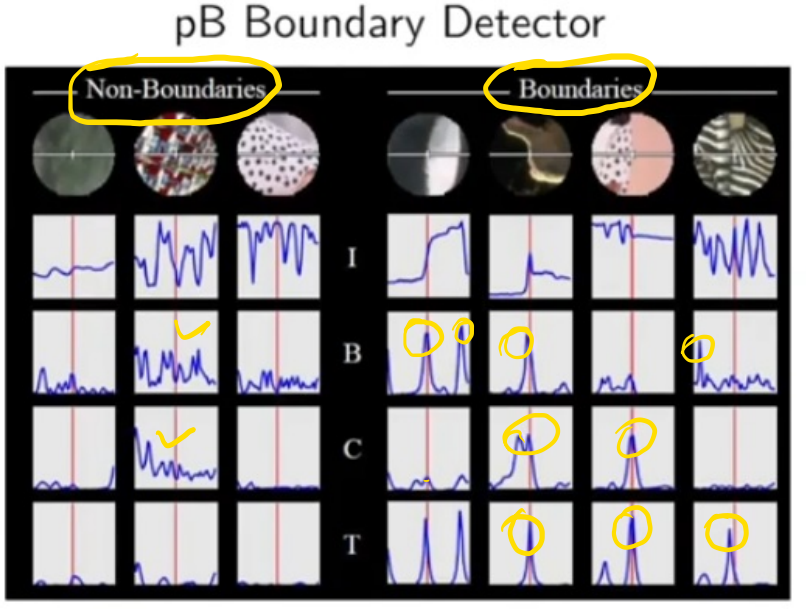
There have been other efforts that people have come up with different other methods, sophisticated methods to do edge detection even until recent years.

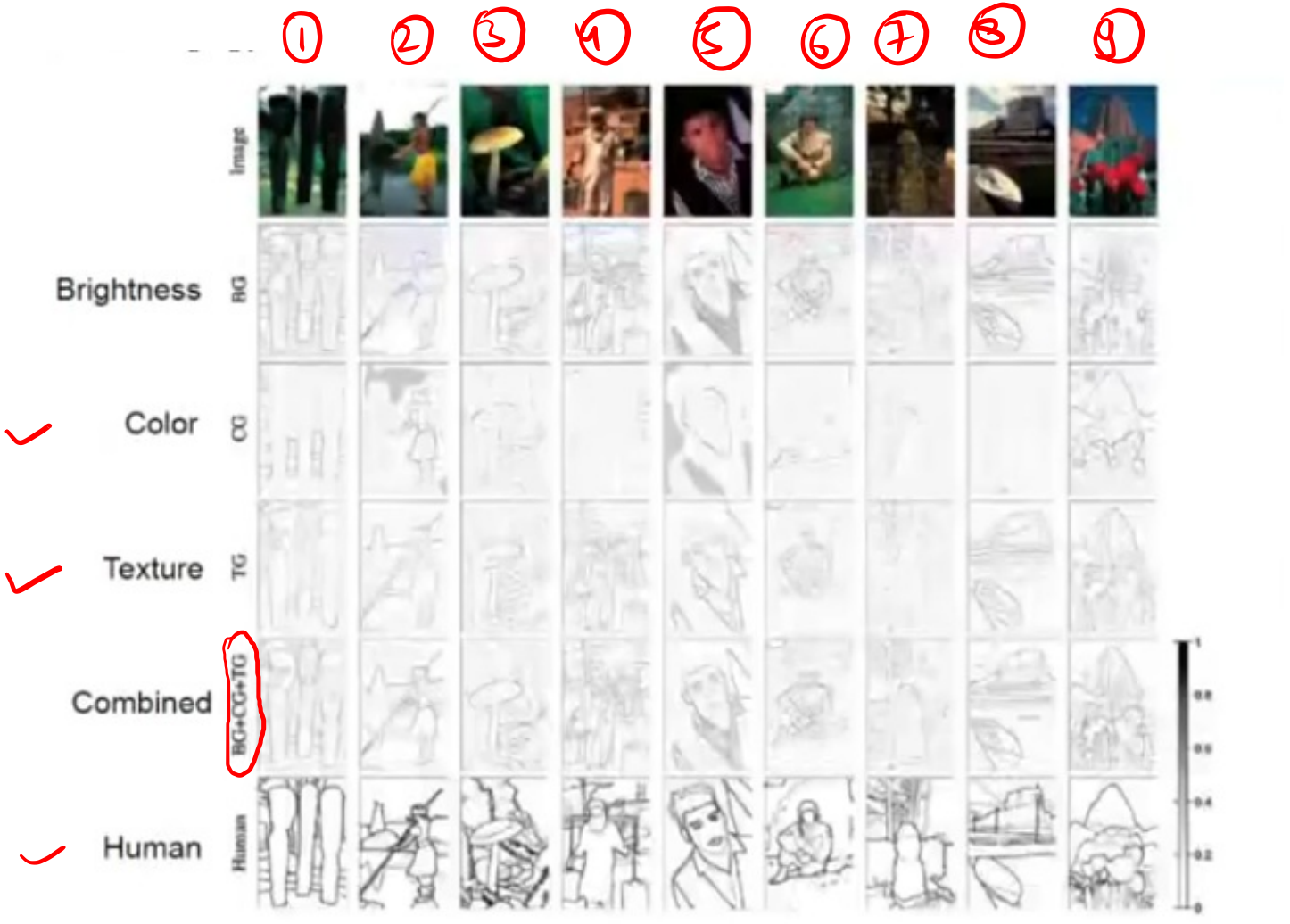
So let us try to visit them at a high level to give you a perspective of how edge detection can be solved using other kinds of approaches.

More Recent Methods in Edge Detection

Learning to detect Natural Image boundaries using local brightness, color, and texture cues (2004, Martin et al.)






1
2
3
4
5
6
7
8
9
Image
Brightness
BG
Color
CG
Texture
TG
Combined
BG+CG+TG
Human
Human
1
0.8
0.6
0.4
0.2
0

Martin and others in 2004, proposed a method called Learning to Detect Natural Image Boundaries Using Local Brightness, Color and Texture Cues. And as the title says, what these group of researchers did is they took an image and they came up with the texture in the image, the brightness in the image and the color in the image

They end up giving this to a classifier, a machine learning based classifier, and they observed that for non-boundary regions, you have a certain set of patterns that you see, whereas for boundary regions, you see a sharp change in the pixel values.

They simply use this idea to provide these texture, brightness and color pixels at every value to a classifier, which tells you whether there is a particular pixel is an edge or not an edge and they then find a way to combine these informations from multiple sources, brightness, color, and texture to give their final edge detection.

Structured Forest for Fast Edge Detection (2013)

-The idea was to use machine learning based methods, to be able to quickly predict whether a pixel is an edge or not.

-The edges can be learned or predicted by looking at past data, you do not need to exactly compute a derivative pixel.

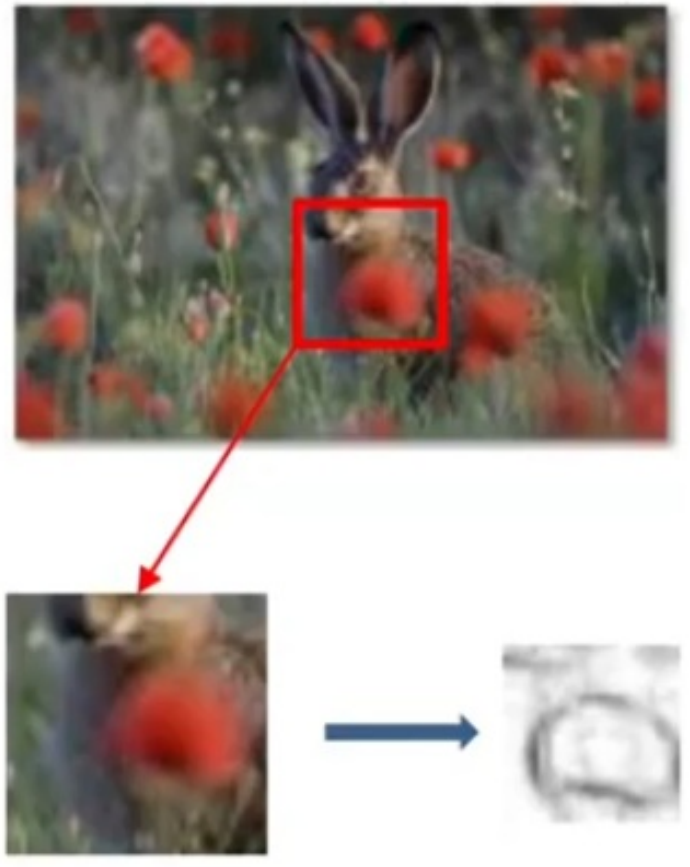
-You can just learn from past data about how edge pixels look, and simply apply that machine learning based classifier to a pixel in this particular image, to give you the outcome of whether that pixel was an edge or not.

-And the other insight is that we want predictions for nearby pixels to be influencing each other. If there are two pixels nearby, if one of them is an edge, there is a good chance that the next pixel is also going to be an edge.

-So these are two insights that they actually use. But the way this method works is it operates at the level of patches in an image and it uses the random forest classifier to be able to tell which pixels in a patch corresponds to an edge.

64X64 32X32 16X16 64X64

32X32



32X32

Let us see this is in a slightly more algorithmic manner.
-You extract a bunch of different overlapping, 32 cross 32 patches at three scales in an image.

-What is a scale? A scale is a resolution in image. So for example, if you had an image whose size was given by 64 cross 64, if you sub-sample it, it becomes 32 cross 32, if sample it still further, it becomes 16 cross 16, these are three different scales of an image.

-So you extract overlapping 32 cross 32 patches at multiples scales.

-And then you consider many features in each of these patches, such as
pixel values
or pairwise differences in colour
or radiant magnitudes
or oriented gradients.

-And using these features, you build a random forest classifier based on some training data where you knew where the edges were.

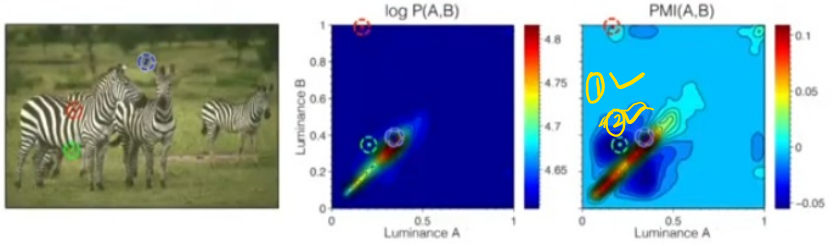
-And in each of these 32 cross 32 patches, you take a central 16 cross 16 region, where you are going to predict the pixel points using these random forest classifiers.

-And you would get such a prediction for the central 16 cross 16 region across many different patches.

-You simply average the predictions for each pixel across all the patches in which that pixel occurred and got a prediction whether that pixel was edge pixel or not an edge pixel.

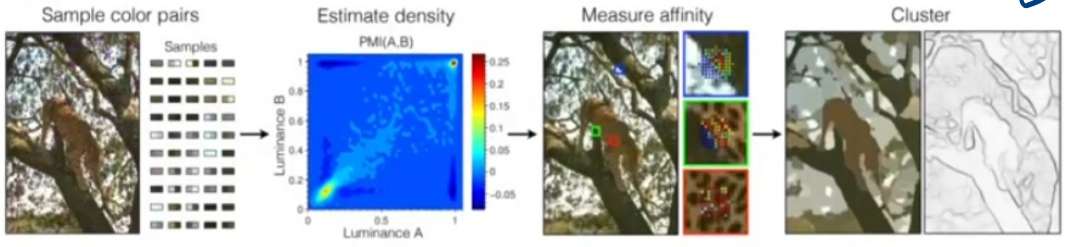
Crisp Boundary Detection using pointwise Mutual Information (2014)

(a) Pixel combinations that are unlikely to be together are edges



$$\mathrm{PMI}_\rho(A,B) = \log \frac{P(A,B)^\rho}{P(A)P(B)}$$

(b) Algorithm Pipeline



This method works as the pixels that are edges share some common information. So they take an image and then they plot all the pairwise co-occurrences of pixels as a density function.

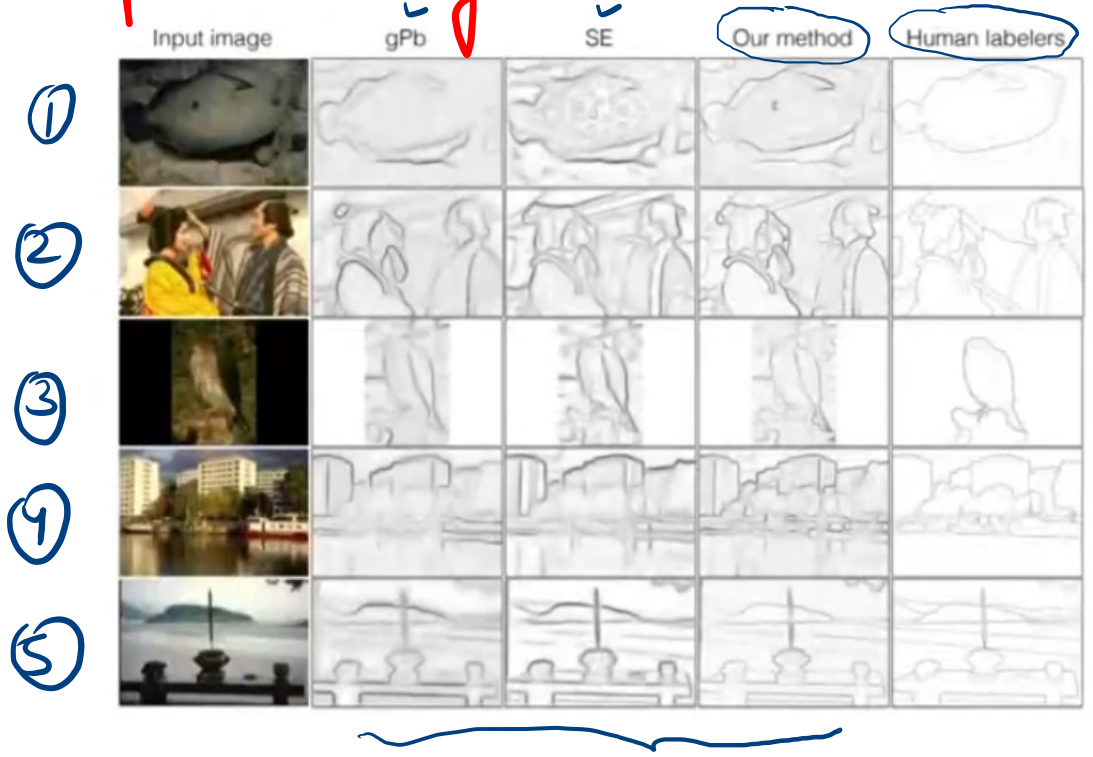
They also compute a quantity called PMI, which is defined as the mutual information in the same image. And then based on these quantities, they come up with something called an affinity matrix, which tells you how close one pixel is to another pixel in terms of how often they co-occur to each other.

And based on that affinity matrix, they use what is known as spectral clustering, which is a clustering method.

In another method, they use spectral clustering to be able to segment and obtain the images in the image.

These are few ideas of how edges can be found through other means.

Crisp Boundary Detection using pointwise Mutual Information (2014)

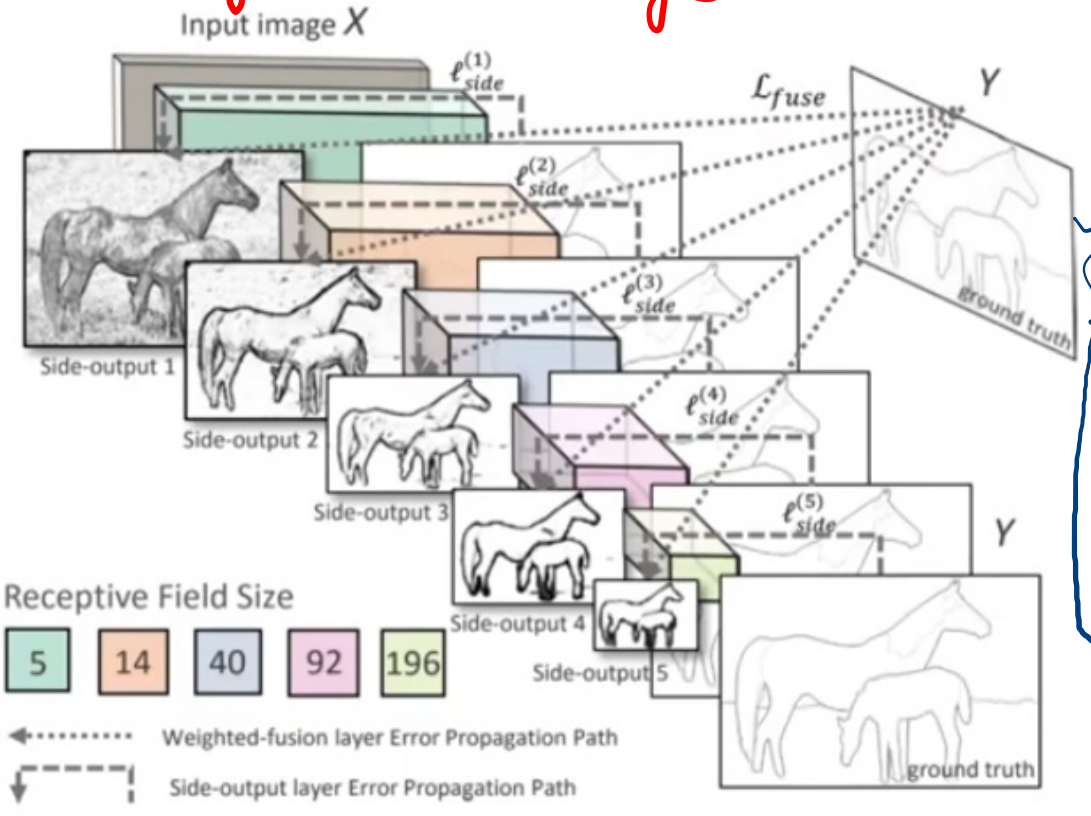


Algorithm	ODS	OIS	AP
Canny [14]	0.60	0.63	0.58
Mean Shift [36]	0.64	0.68	0.56
NCuts [37]	0.64	0.68	0.45
Felz-Hutt [38]	0.61	0.64	0.56
gPb [1]	0.71	0.74	0.65
gPb-owt-ucm [1]	0.73	0.76	0.73
SCG [9]	**0.74**	0.76	0.77
Sketch Tokens [7]	0.73	0.75	0.78
SE [8]	**0.74**	0.76	0.78
Our method – SS, color only	0.72	0.75	0.77
Our method – SS	0.73	0.76	**0.79**
Our method – MS	**0.74**	**0.77**	0.78

Evaluation on BSDS500

This is an example where they show that given these input images, their methods give fairly good edges, which are very similar to how human labelers give outputs for the same edges.

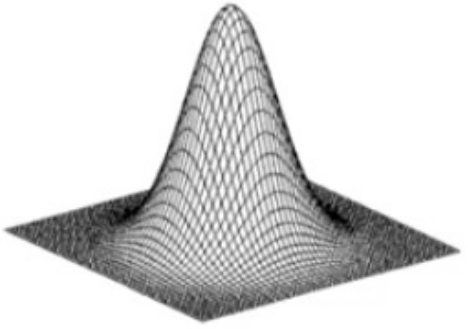
Holistically Nested Edge Detection



	ODS	OIS	AP	FPS
Human	.80	.80	-	-
Canny	.600	.640	.580	15
Felz-Hutt [9]	.610	.640	.560	10
BEL [5]	.660+	-	-	1/10
gPb-owt-ucm [1]	.726	.757	.696	1/240
Sketch Tokens [24]	.727	.746	.780	1
SCG [31]	.739	.758	.773	1/280
SE-Var [6]	.746	.767	.803	2.5
OEF [13]	.749	.772	.817	-
DeepNets [21]	.738	.759	.758	1/5†
N4-Fields [10]	.753	.769	.784	1/6†
DeepEdge [2]	.753	.772	.807	$1/10^3$†
CSCNN [19]	.756	.775	.798	-
DeepContour [34]	.756	.773	.797	1/30†
HED (ours)	**.782**	**.804**	**.833**	2.5†, 1/12

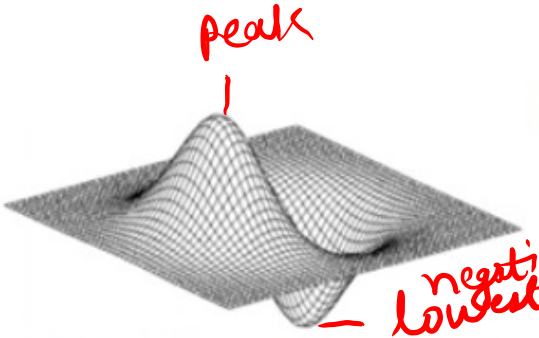
This is a deep learning based method. This method actually uses convolutional neural networks to find edges, where you can see given an input, these are what are known as convolutional layers. And at the end of every convolutional layer, you try to predict the edges in the image.

Detection of blobs



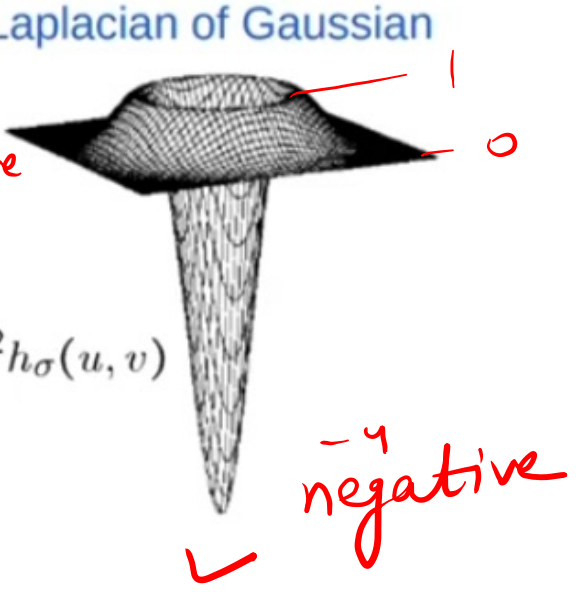
Gaussian

$$h_\sigma(u,v) = \frac{1}{2\pi\sigma^2} e^{-\frac{u^2+v^2}{2\sigma^2}}$$



Derivative of Gaussian

$$\frac{\partial}{\partial x} h_\sigma(u,v)$$



$$\nabla^2 h_\sigma(u,v)$$

Laplacian $\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 y}{\partial y^2}$

Laplacian of Gaussian

Example

3×3 Laplacian of Gaussian filter

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

How did we obtain this filter?

Discrete approximation of second derivative

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

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

Substituting these values in the Log equation

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

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

Converting this equation to the filter results in the given LoG matrix.

clarity

Original image

Laplacian

Laplacian of Gaussian

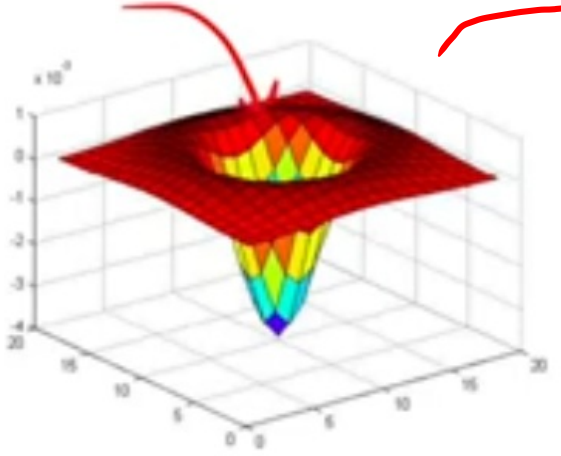
Laplacian of Gaussian gives you smoothen effect which smoothen out the noise and then take the Laplacian of original image.

What else can Laplacian of Gaussian can do?

What else can Laplacian of Gaussian can do?

- In other thing, LoG can detect blobs.

LOG as a blob detector

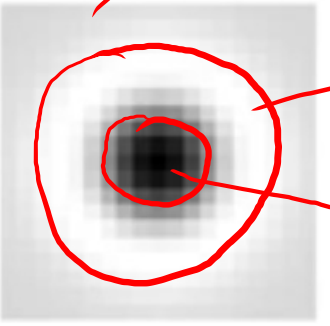


LOG

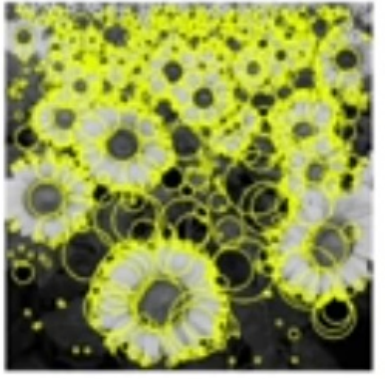
gray shades

white

black hole



LOG as an image



Convolution with a filter can be viewed as a little picture of what you want to find against all local regions in the image.

LOG filter matrix reveals that it is circularly symmetric. Thus it can be used for blob detection.

For the edge detection or similar detection of other artifacts, they only take the absolute value, whether the output is negative or positive.

That would change if you had a white circle with a black whole or black circle with white whole.

Ideally both of them are recognized as blobs

So Laplacian of Gaussian (LOG) central peak may goes up on top or comes down below.

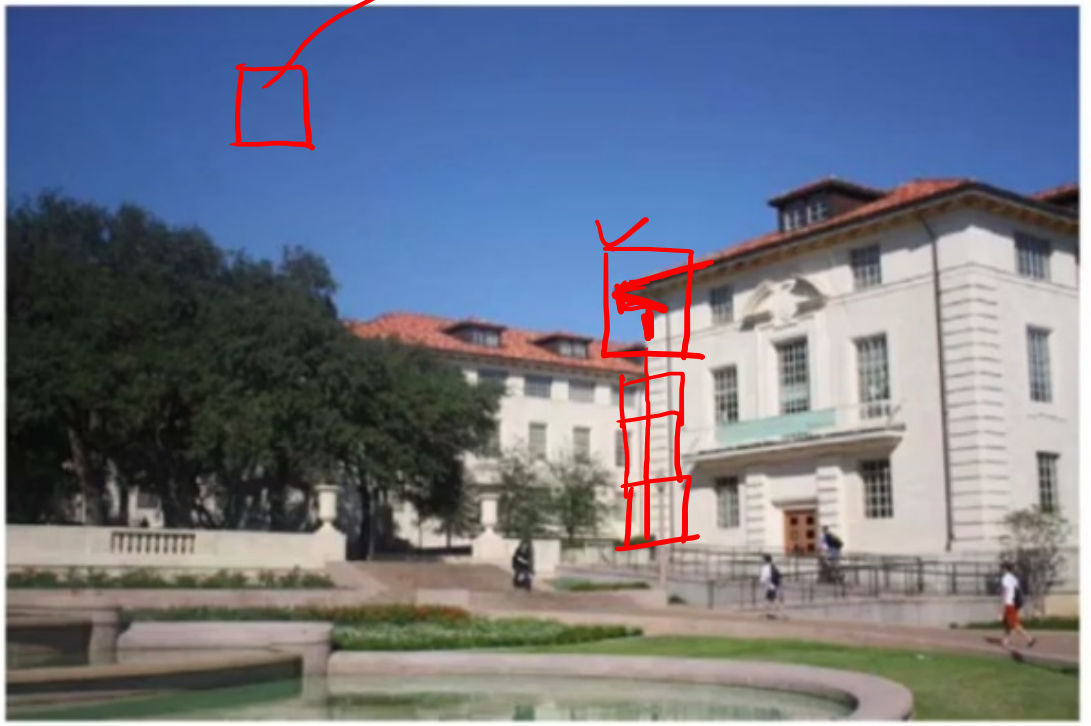
If you take a Laplacian of Gaussian filter, 3x3 filter size is too small. You may take a 7x7 or 11x11 LOG filter.

You can visualize it as black blob with circled around white ring and rest of the places are gray. You may have invert of it as well.

With LoG, you can count sunflowers in a field or you can detect red blood cells in your blood test or any structure with blobs across the image.

In the image of sunflower, there are blobs of different sizes. So you would have to run LoG with different blob size to capture all of those blobs present in the image.

Detection of Corners



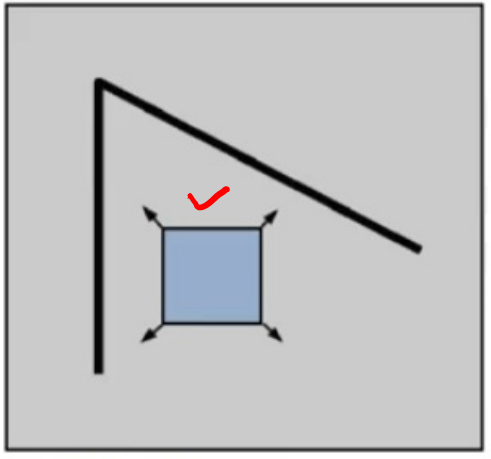
Extract the features of this image in such a way that I can differentiate this image with another image.

Corner detection

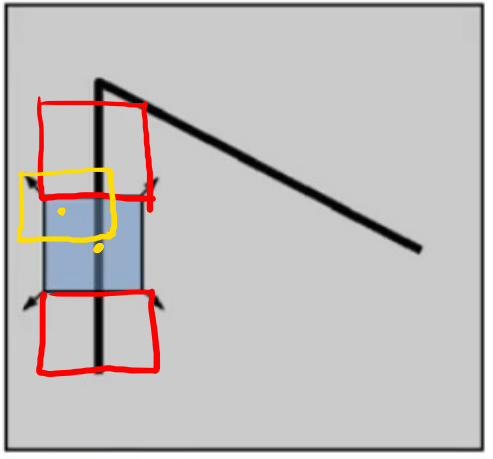
- Look for image regions that are unusual. How to define "Unusual"?
- Textureless patches are nearly impossible to localize.
- Patches with large contrast changes (gradients) are easier to localize.
- But straight line segments at a single orientation suffer from the aperture problem i.e. it is only possible to align the patches along the direction normal to the edge direction.

- Gradients in at least two (significantly) different orientations are the easiest, e.g., corners.

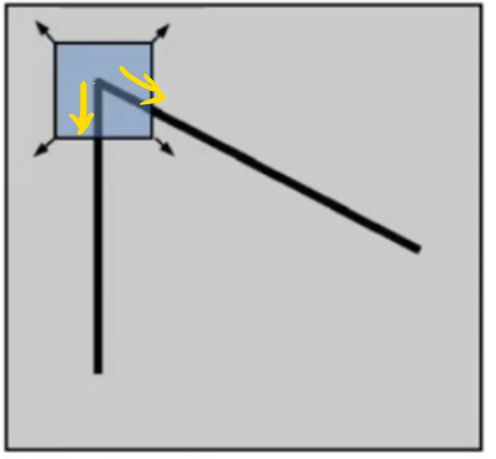
Consider a small window of pixels. How does the window change when you shift it



"flat" region:
no change in all directions



"edge":
no change along the edge direction



"corner":
significant change in all directions

In this example, we have inverted 'V' as an image.

We want to find out which aspect of the image is unique to it, which can help us to recognize it.

Now we will see the image from different angles. If you consider a blue box placed there in each case at different locations, to identify flat region, edge and corner.

① In the first case, you see the flat textureless region. There is no change in the intensity in that region. It is like the blue sky with absolutely no change.

This is not going to be very useful when you try to compare with other images.

(2) In case 2, if you place the blue box on the edge part of the image. But the problem is you never know where to place the blue box you can place it anywhere on the vertical line

Because all of them would have the same response. There is no difference in the local characteristics in all of these places.

③ In case 3, we are ideally looking for placing the box where there is change in two directions and that point could be unique to this particular image.

Ideally we are looking for many such points in an image.

How do you identify such points in an image?

Auto correlation

- It is correlation with itself. So you are not going to use external filter.
- Auto-correlation function compute the sum of squared differences between pixel intensities with respect to small variations in the image patch position.

$$E_{AC}(\underset{\downarrow}{\Delta u}) = \sum_{x,y} w(P_i) \left[I(P_i + \delta u) - I(P_i) \right]^2$$

vector

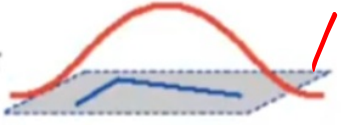
$\delta u, \delta v$

where $P_i = (x, y)$, a particular position on the image.

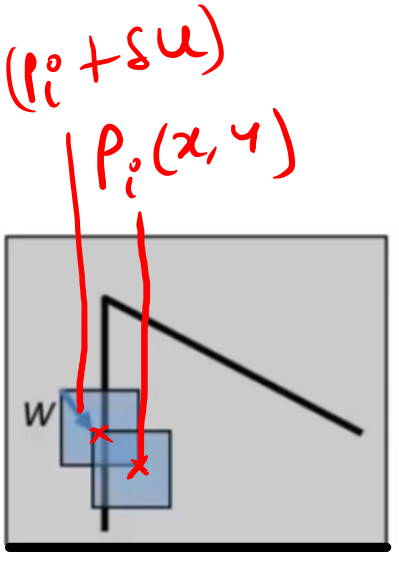
Window function $w(x,y)$ =

1 in window, 0 outside

or



Gaussian



fixed weight $\omega(P_i)$

weight $\omega(P_i)$

using a Taylor Series expansion

$$I(p_i + \Delta u) = I(p_i) + \nabla I(p_i)\,\Delta u$$

$p_i + \Delta u$ is about taking a patch which was centered at p_i and moving it by a delta u and placing it slightly offset location in the same image.

with the image gradient

$$\nabla I(p_i) = \left(\frac{\delta I(p_i)}{\delta x}, \frac{\delta I(p_i)}{\delta y} \right)$$

Autocorrelation can be approximated as:

$$E_{AC}(\Delta u) = \sum_{x,y} W(P_i)\left[I(P_i+\delta u) - I(P_i)\right]^2$$

$$\approx \sum_{x,y} W(P_i)\left[I(P_i) + \delta I(P_i)\delta u - I(P_i)\right]^2$$

$$= \sum_{x,y} W(P_i)\left[\delta I(P_i)\,\delta u\right]^2$$

$$= \Delta u^T A \Delta u$$

Autocorrelation can be approximated as:

$$E_{AC}(\Delta u) = \sum_{x,y} W(P_i)\left[I(P_i+\delta u) - I(P_i)\right]^2$$

$$\approx \sum_{x,y} W(P_i)\left[I(P_i) + \delta I(P_i)\delta u - I(P_i)\right]^2$$

$$= \sum_{x,y} W(P_i)\left[\delta I(P_i)\delta u\right]^2$$

$$= \Delta u^T A \Delta u$$

The autocorrelation is $E_{AC}(\Delta u) = \Delta u^T A \, \Delta u$

$\forall I(P_i)$

with $A = \sum_u \sum_v w(u,v) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$

$$= w * \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

* The weighted summation have been replaced with discrete convolutions with the weighting kernel w.

* The eigenvalues of A reveal the amount of intensity change in the two principle orthogonal gradient directions in the window.

2D

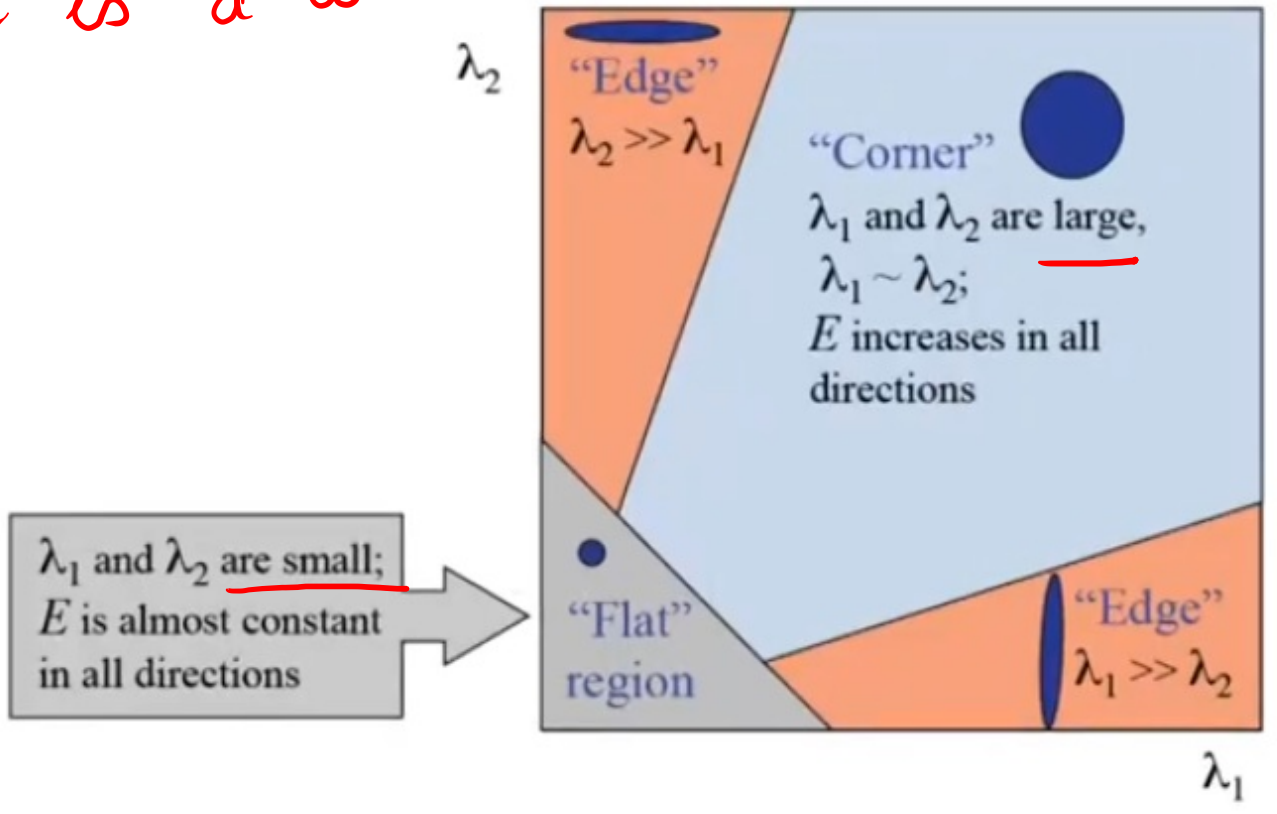
$$A = U \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}_{2\times 2} U^T$$

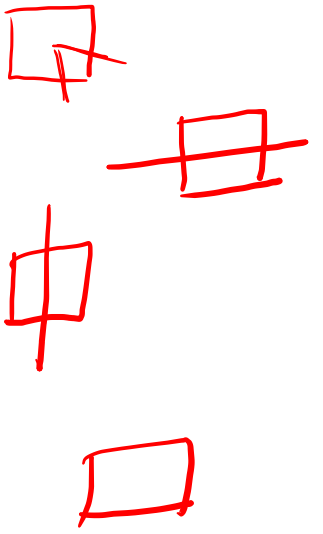
with $Au_i = \lambda u_i$

Diagonal

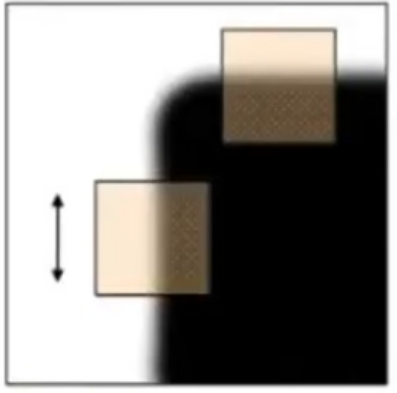
eigen value decomposition

How do eigen values determine if an image point is a corner?





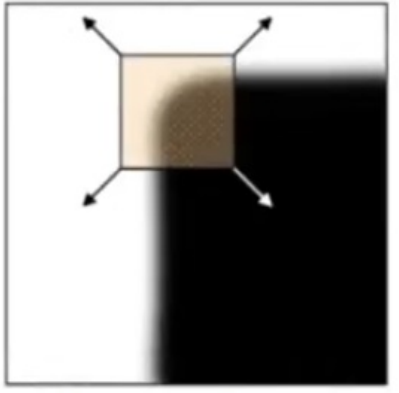
① Wherever there is a vertical or horizontal edge, you are going to have either λ_1 greater than λ_2 or λ_2 greater than λ_1.



"edge":

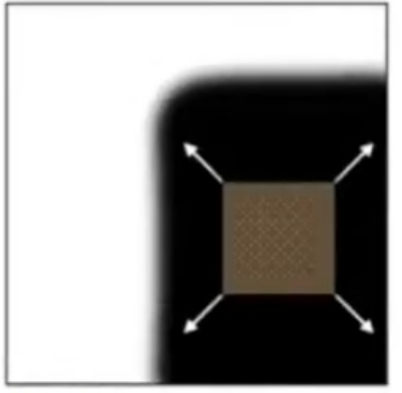
$\lambda_1 >> \lambda_2$

$\lambda_2 >> \lambda_1$



"corner":

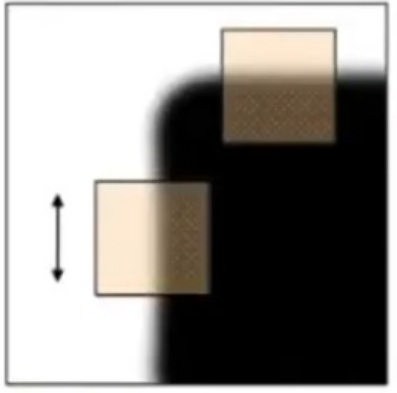
λ_1 and λ_2 are large, $\lambda_1 \sim \lambda_2$;



"flat" region

λ_1 and λ_2 are small;

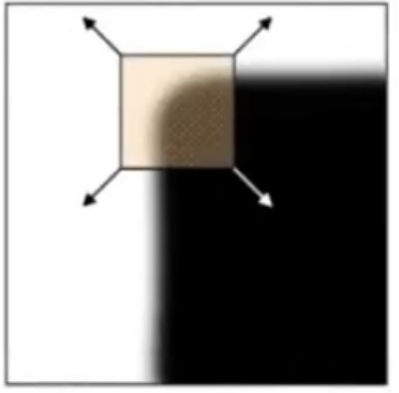
① Wherever there is a vertical or horizontal edge, you are going to have either λ_1 greater than λ_2 or λ_2 greater than λ_1.



"edge":

$\lambda_1 >> \lambda_2$

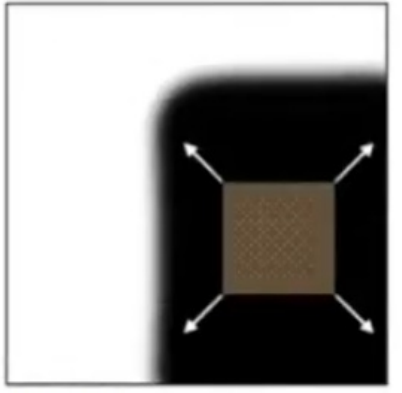
$\lambda_2 >> \lambda_1$



"corner":

λ_1 and λ_2 are large,

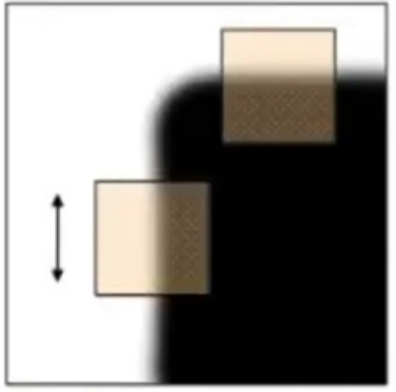
$\lambda_1 \sim \lambda_2$;



"flat" region

λ_1 and λ_2 are small;

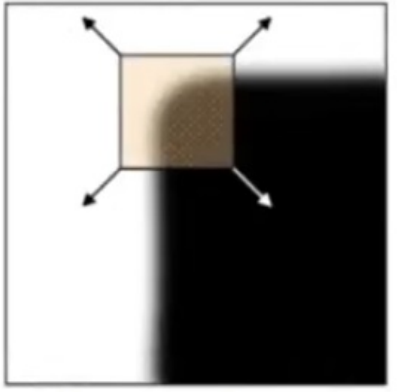
② At corner, we are going to have both λ_1 and λ_2 to be large.



"edge":

$\lambda_1 >> \lambda_2$

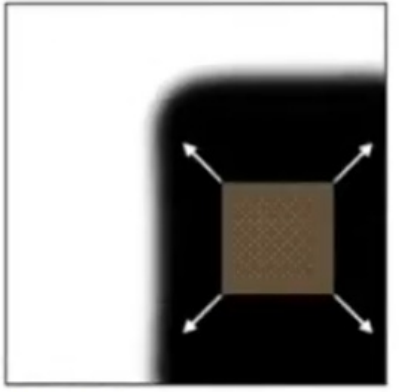
$\lambda_2 >> \lambda_1$



"corner":

λ_1 and λ_2 are large,

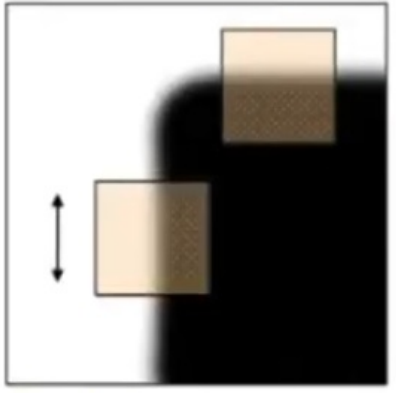
$\lambda_1 \sim \lambda_2$;



"flat" region

λ_1 and λ_2 are small;

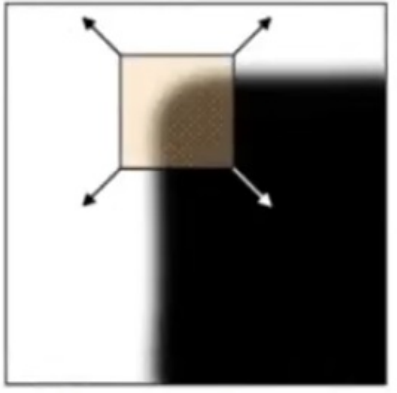
③ In a flat region, you are going to have both λ_1 and λ_2 to be very small.



"edge":

$\lambda_1 >> \lambda_2$

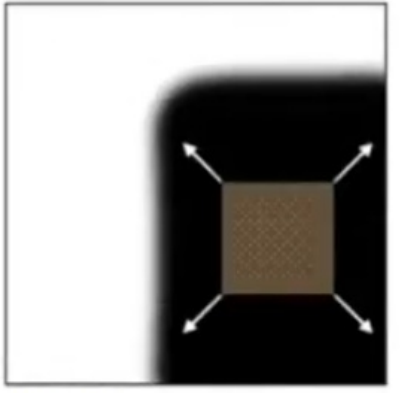
$\lambda_2 >> \lambda_1$



"corner":

λ_1 and λ_2 are large,

$\lambda_1 \sim \lambda_2$;



"flat" region

λ_1 and λ_2 are small;

Harris Corner Detector

The way we are computing corners, basically it is a method given by Harris. This is very popular from several years.

The following steps are followed in this method:

- Compute gradients at each point in the image.

  using it, you can compute matrix A

- Compute A for each image window to get its correctness scores.

- Compute the eigen values:

$$M_c = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$$

$$= \det(A) - k\,\text{trace}^2(A)$$

k → Constant

1+2+1 = 4

* The computation of eigen values is a costly process.
* When the entire quantity is high, you will know that both those λ_1 and λ_2 are high.
* We can define the cornerness measure as determinant of A minus k times trace² of A

* for different images you have to set k differently.
* The computation of determinant and trace is much more easier than computation of eigen decomposition.

- Find points whose surrounding window gave larger cornerness response (M_c > threshold)
- Take points of local maxima, perform non maximum Suppression.

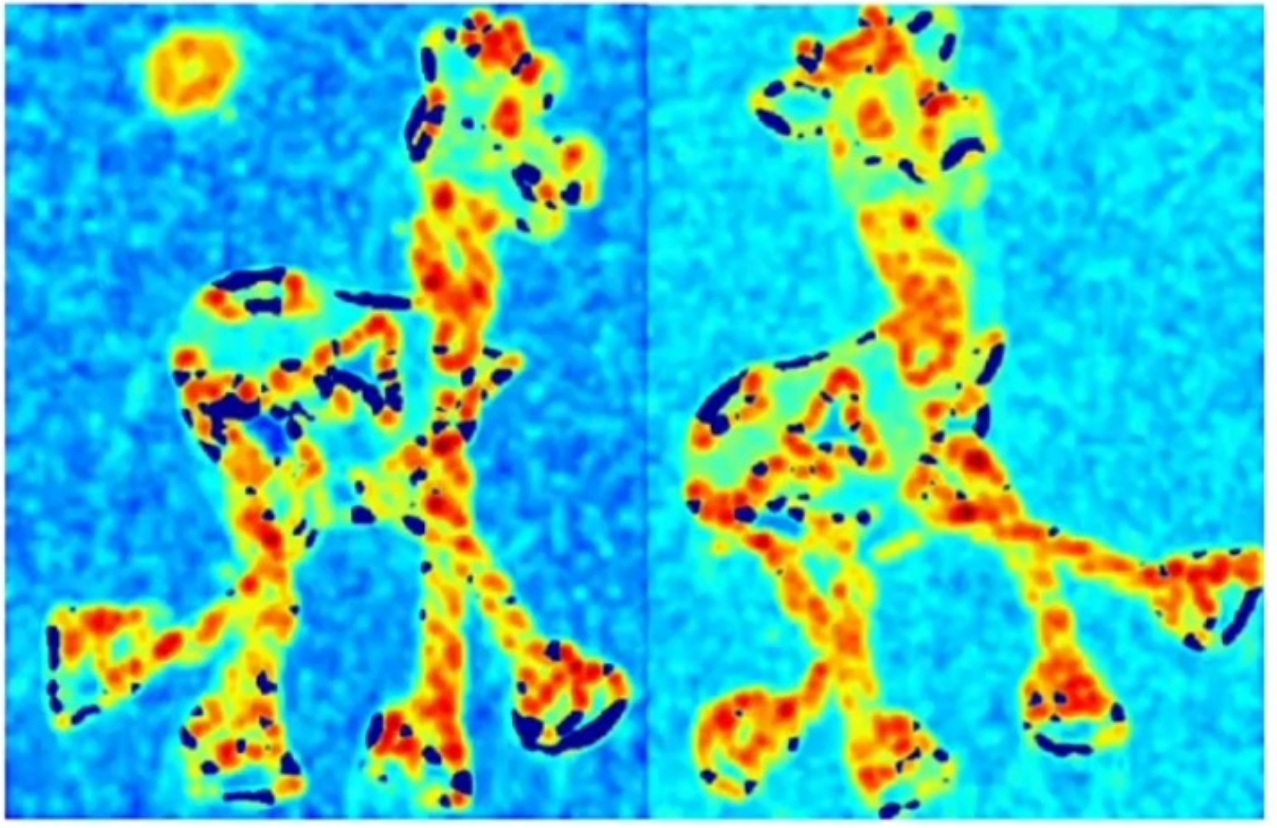
Example

Same objects taken from different angles

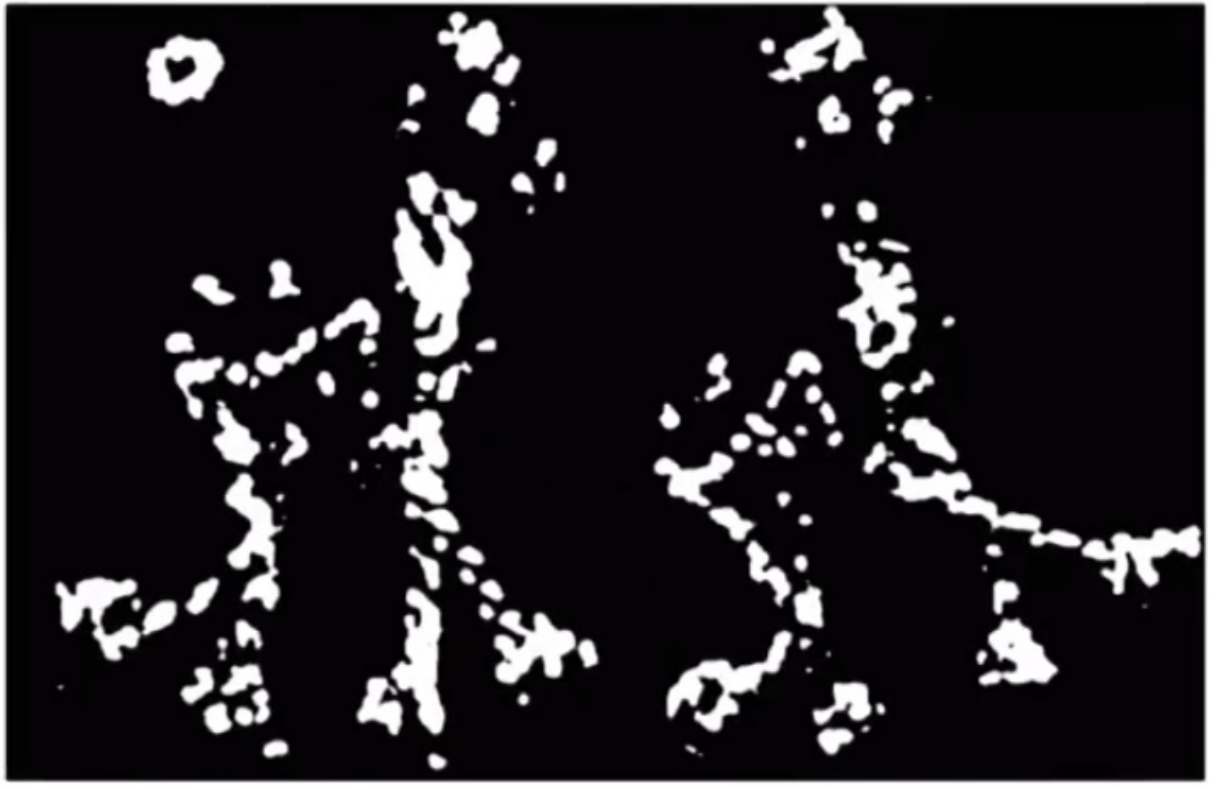
The objects are at different poses and the illumination is also different.

In both the objects same corners should be picked

Through auto correlation compute cornerness values

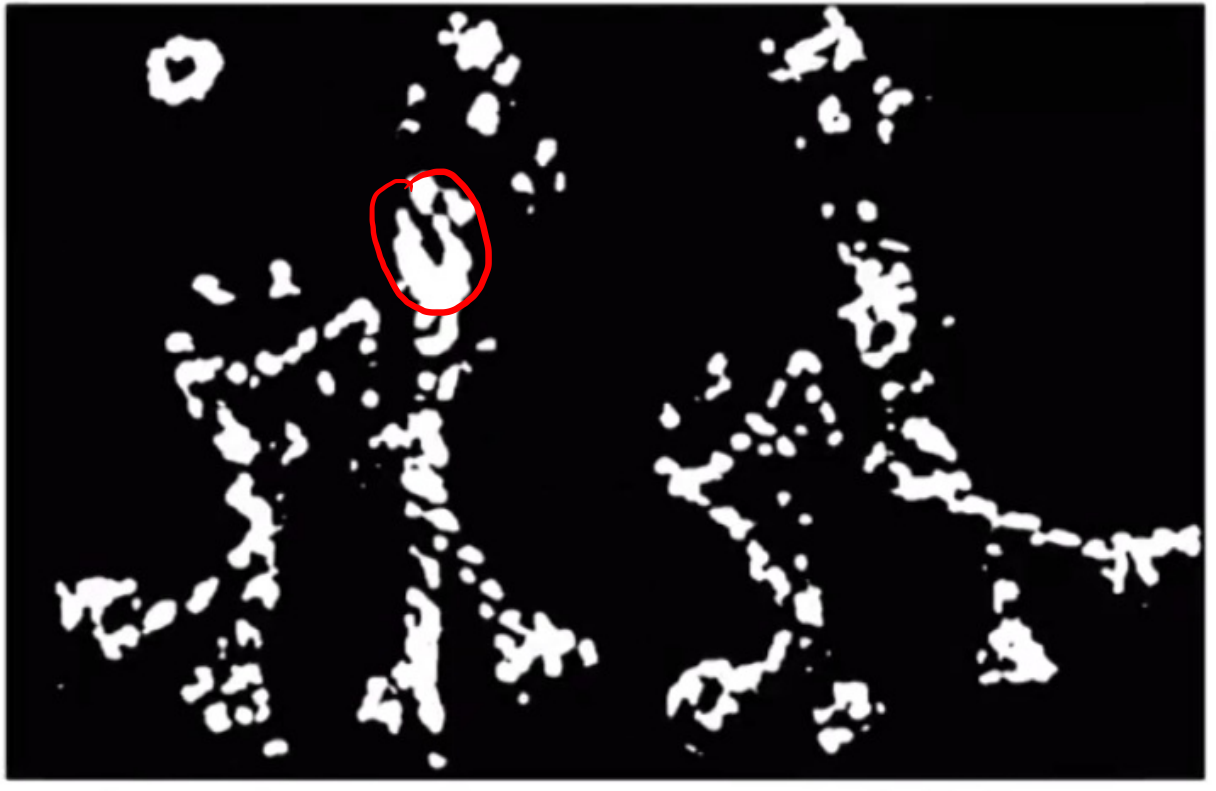


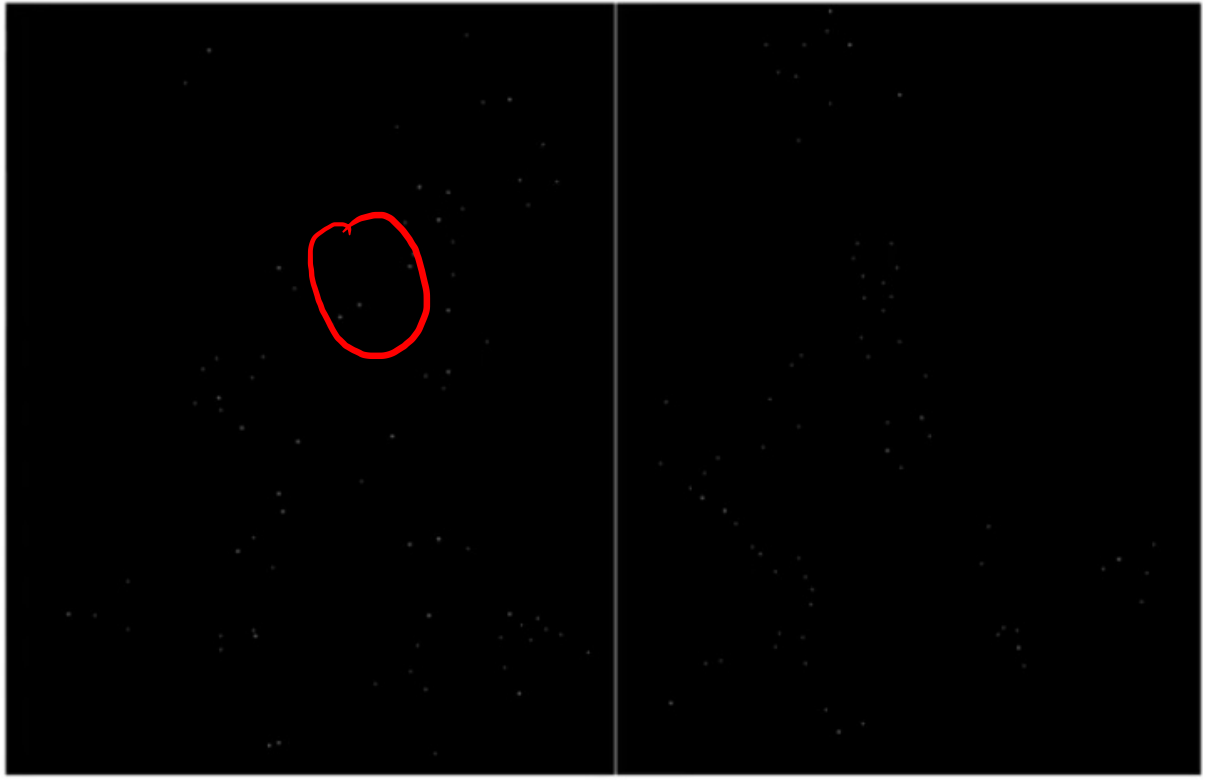
Then you take all high responses while using a threshold.

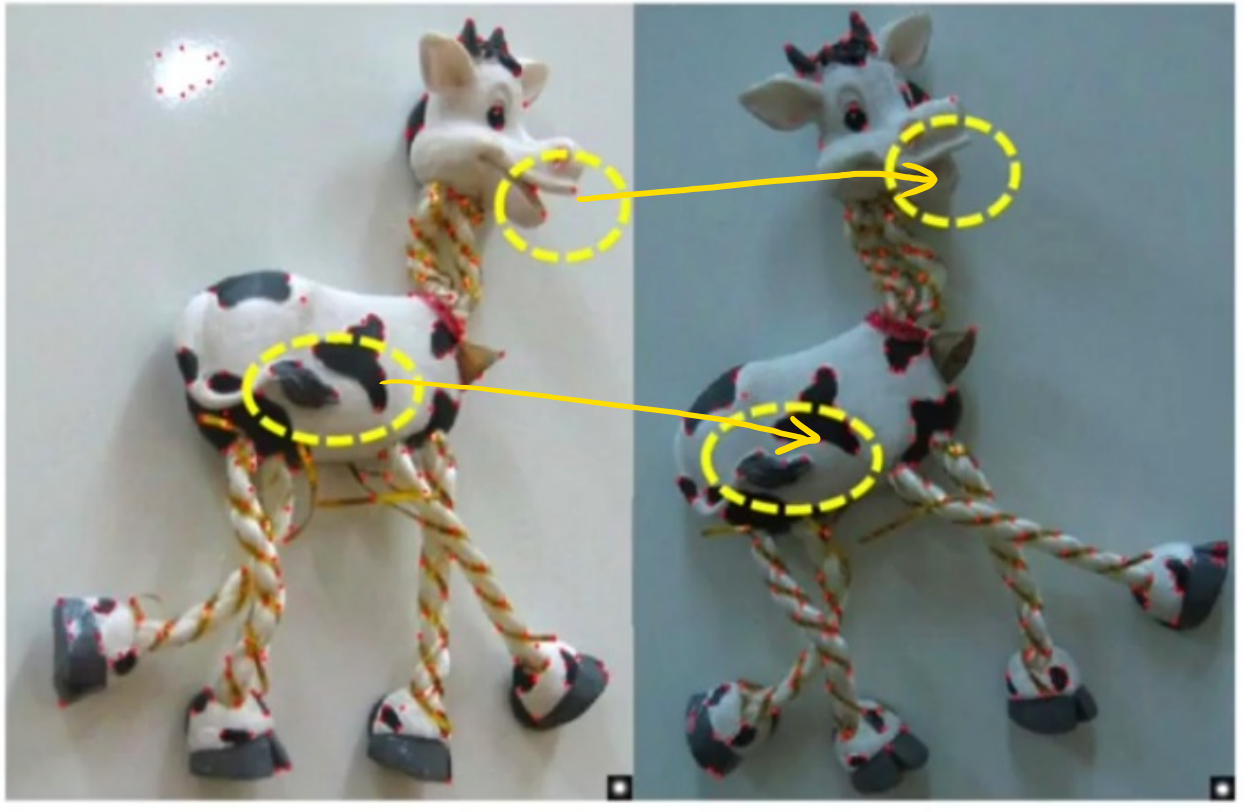


Due to non-maximum supression, you see many of those points in regions

You will see bunch of points in those regions







Harris Corner Detector: Variants

- Harris and Stephens' 88 is rotationally invariant and downweights edge-like features where $\lambda_1 >> \lambda_0$

$$\det(A) - \alpha \, \text{trace}(A)^2 = \lambda_0 \lambda_1 - \alpha (\lambda_0 + \lambda_1)^2$$

- Triggs'04 Suggested

$$\lambda_0 - \alpha \lambda_1$$

- Brown et al, '05 use harmonic mean:

$$\frac{\det(A)}{\text{trace}(A)} = \frac{\lambda_0 \lambda_1}{\lambda_0 + \lambda_1}$$

(which is smoother when $\lambda_0 \approx \lambda_1$)

How do you go from canny edges to straight lines?
Remember canny edges will find, because you use gradient magnitude, canny edges will give you edges in any orientation.

But how do you find the straight lines and what the parameterization of the straight line will be?
Think about it