Imperial College London

8 – Feature Detection

Prof Peter YK Cheung

Dyson School of Design Engineering

URL: www.ee.ic.ac.uk/pcheung/teaching/DE4_DVS/
E-mail: p.cheung@imperial.ac.uk

The materials in this Lecture is based on the first half of Chapter 10 of the recommended textbook, “Digital Image Processing” by Gonzalez and Woods.

This lecture is all about detecting features, particularly boundaries, lines and edges. The technique described here are based on two most important ideas: **derivatives** and **gaussian function**. Features are usually found by discontinuity in intensity, which derivatives help to find. However derivatives has the tendency to emphasize noise, thus creating artifacts and spurious features. Gaussian function is important in reducing these negative aspects of derivatives by smoothing noise and abrupt changes.

What is meant by "discontinuity"?

- Discontinuity in intensity is normally identified by the 1st order and 2nd order derivatives (lecture 5 slides 13, 14).
- We use central difference to compute the 1st order derivative as:

$$\frac{\partial f(x)}{\partial x} = f'(x) = \frac{f(x+1) - f(x-1)}{2}$$

- The 2nd order derivative is given by:

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

- We rarely use 3rd order derivatives. Nevertheless, here it is just for information:

$$\frac{\partial f^3(x)}{\partial x^3} = f'''(x) = \frac{f(x+2) - 2f(x+1) + 2f(x-1) - f(x-2)}{2}$$

We have already covered first- and second-order derivatives in Lecture 5 slides 13 and 14 briefly. In 1-D case, i.e. for a row of pixels, first order derivative is calculated using the so-call difference equation: i.e. finding differences between neighbourhood pixels.

If the current 1D pixel is at location x, then first order derivative is calculated by the difference between the two neighbours at x+1 and x-1. Hence the equation is (f(x) is the intensity value at location x:

$$\frac{\partial f(x)}{\partial x} = f'(x) = \frac{f(x+1) - f(x-1)}{2}$$

Similar, for 2nd order derivative, the equation is given by:

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

We very rarely use 3rd order derivatives. It is given here for completeness.

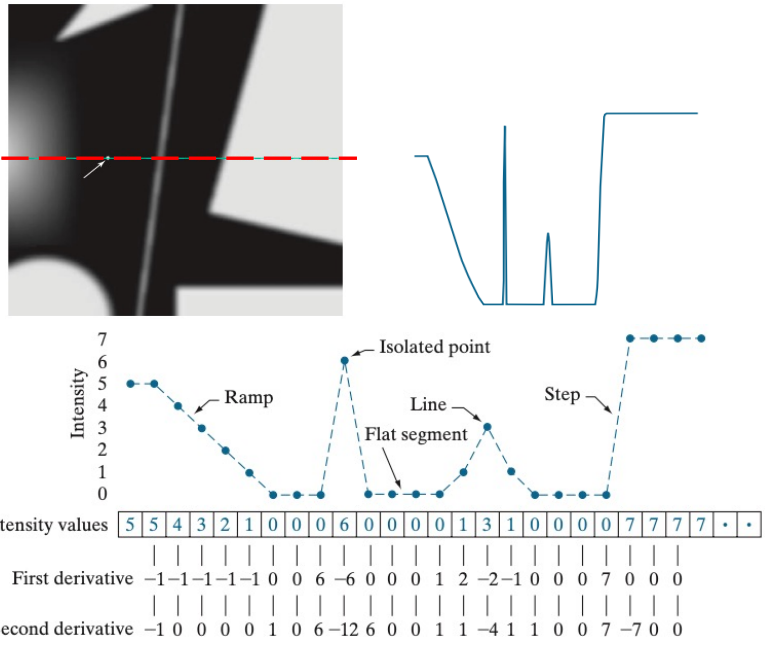
Digital Derivatives - coefficients

- To generalise, here is a table of the first four central digital derivatives coefficients:

	$f(x+2)$	$f(x+1)$	$f(x)$	$f(x-1)$	$f(x-2)$
$2f'(x)$		1	0	−1	
$f''(x)$		1	−2	1	
$2f'''(x)$	1	−2	0	2	−1
$f''''(x)$	1	−4	6	−4	1

Here are the coefficients (i.e. constant multipliers for neighbouring pixel intensities) for calculating the central digital derivatives.

Cross section of an image & derivatives



Just to illustrate how image intensity affects derivatives, this slide shows an image and a cross-section along the red line. The intensity values are plotted on the right.

Below is a diagram showing which is the ramp, point, line and step features at the cross-section.

Below are the first and second derivatives showing what results these will provide.

As can be seen, it is rather easy to spot the point, line and step using these derivative values.

Detection of Isolated point

- The obvious approach is to perform spatial filtering with a kernel that compute the 2nd order derivative (also called the Laplacian):

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{dx^2} + \frac{\partial^2 f}{dy^2}$$
$$= f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y)$$

- This is equivalent to performing convolution with the filter kernel, but negate the output:

0	-1	0
-1	4	-1
0	-1	0

or

-1	-1	-1
-1	8	-1
-1	-1	-1

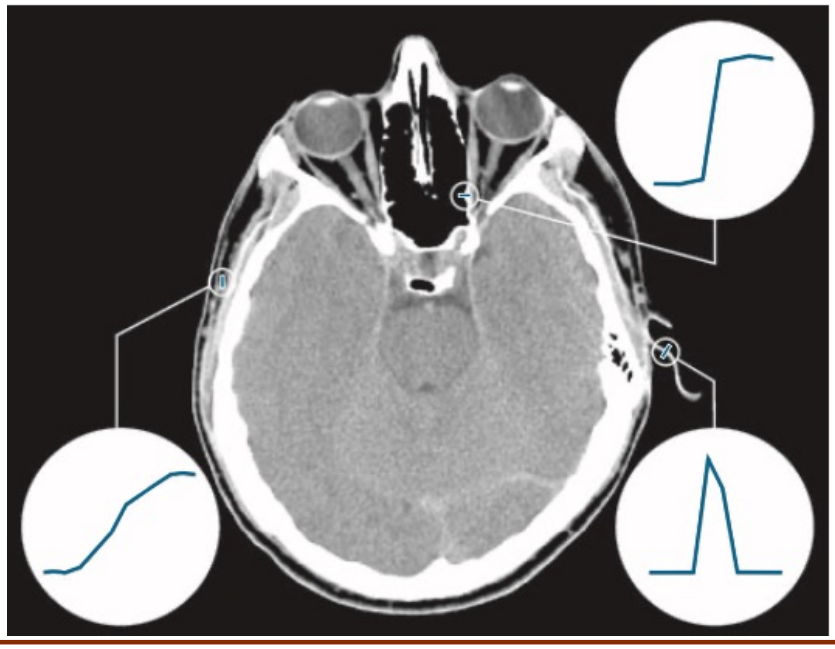
To detect a point, we can perform Laplacian at (x,y), i.e. calculate the following:

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{dx^2} + \frac{\partial^2 f}{dy^2} = f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y)$$

For uniform intensity area, the value is 0. However, for a point intensity, this will compute to a large negative value. So, we can detect a point by perform spatial filtering, or convolution of the image with the filter kernel as shown on the left. The coefficients are negated so that a point will give a large positive value.

Alternatively and even better, we can use the kernel shown on the right, which detect a point discontinuity in all eight directions (x, y and two diagonals).

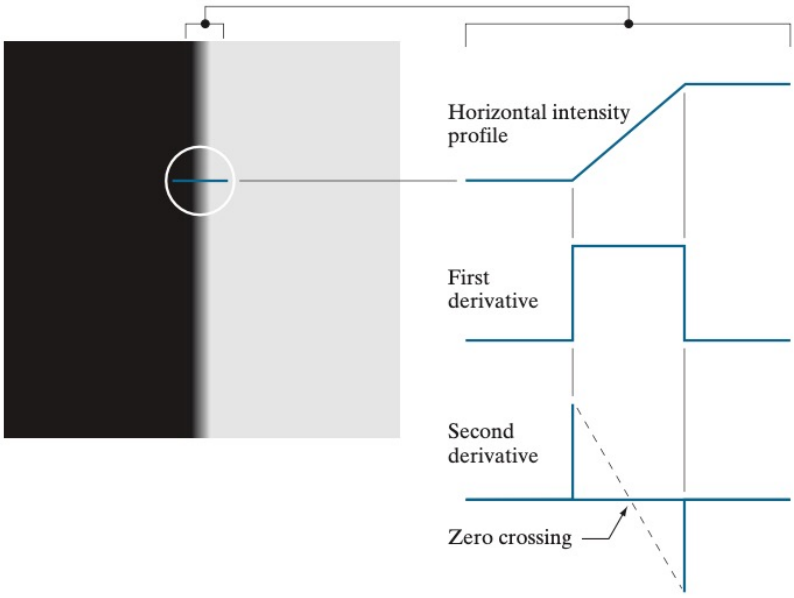
Three types of edges



Next type of feature would have been a line. However, we will jump straight to edges because one can consider edges are general lines. For example, a line that is more than one pixel wide is effective two steps or one roof edge.

The slide above shows three types of edges: a step, a ramp and a roof edge. These three types can be found in a typical image such as the X-ray image of a head. The three types of edges are identified.

Edge detection using derivatives



The first derivative can be used to detect the presence of an edge at a point in an image. Similarly, the sign of the second derivative can be used to determine whether an edge pixel lies on the dark or light side of an edge.

Two additional properties of the second derivative around an edge are:

(1) it produces two values for every edge in an image; and

(2) Its zero crossings can be used for locating the centres of thick edges.

Some edge models utilize a smooth transition into and out of the ramp. Finally, although attention thus far has been limited to a 1-D horizontal profile, a similar argument applies to an edge of any orientation in an image. We simply define a profile perpendicular to the edge direction at any desired point and interpret the results in the same manner as for the vertical edge just discussed.

Sobel Edge detector kernels

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

−1	−2	−1
0	0	0
1	2	1

$$g_x = \frac{\partial f}{dx} = (w_7 + 2w_8 + w_9) - (w_1 + 2w_2 + w_3)$$

−1	0	1
−2	0	2
−1	0	1

$$g_y = \frac{\partial f}{dy} = (w_3 + 2w_6 + w_9) - (w_1 + 2w_4 + w_7)$$

The first derivative can be used to detect the presence of an edge at a point in a given direction such as x and y. Shown in this slide are two kernels known as **Sobel edge detector**. The top kernel detects horizontal edges while the second kernel detects vertical edges. These are both based on unidirectional 1st derivatives.

Sobel edge detector has many limitations. It is not isotropic meaning that it is sensitive to the direction of the feature. It is also not adjustable to feature size. For single pixel edge, it works reasonably well. However for thicker edges, this does not work at all. So we need something that can be adjust to the size of the feature.

Laplacian of a Gaussian (LoG) edge detector (1)

- Marr-Hildreth proposed an edge detector which is has two properties:
  1. Compute 1st or 2nd derivative at every point in the image
  2. Capable of being "tuned" to any scale or size
- The operator they proposed is the **Laplacian** (or 2nd derivative) **of** a **Gaussian** function.
- A 2D Gaussian function is defined as:

$$G(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}}$$

- The Laplacian of a Gaussian (LoG) is defined as:

$$\nabla^2 G(x,y) = \left(\frac{x^2+y^2-2\sigma^2}{\sigma^4}\right) e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The first derivative are image scale dependent. This means that detection of edges using derivatives alone requires using operators of different sizes, and a sudden intensity change will give rise to a peak or trough in the first derivative and a zero crossing in the second derivative. That is why Sobel edge detectors do not work well.

A good edge detection should have two salient features:

1) it should be a differential operator capable of computing a digital approximation of the first or second derivative at every point in the image;
2) It should be capable of being "tuned" to act at any desired scale, so that large operators can be used to detect blurry edges and small operators to detect sharply focused fine detail.

The most appropriate operator fulfilling these conditions is the 2-D Gaussian filter function given by the filter $\nabla^2 G$, where G is:

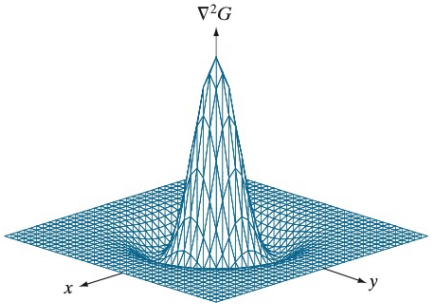
$$G(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}}$$

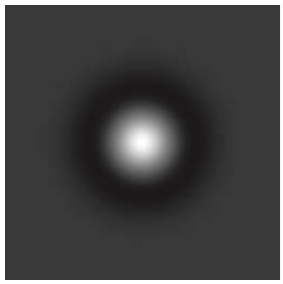
with σ being the standard deviation (or called space constant) adjustable to determine the size of the operator.

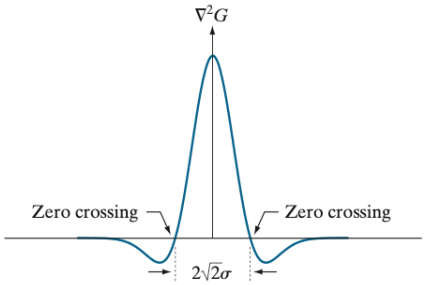
By performing differentiation, it can be shown that the Laplacian of a Gaussian (LoG) is give by:

$$\nabla^2 G(x,y) = \left(\frac{x^2+y^2-2\sigma^2}{\sigma^4}\right) e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Laplacian of a Gaussian (LoG) edge detector (2)







0	0	−1	0	0
0	−1	−2	−1	0
−1	−2	16	−2	−1
0	−1	−2	−1	0
0	0	−1	0	0

The top-left figure in the slide is a 3-D plot of the negative of the LoG function..

The image on the top-right is the image of the 3-D plot.

The plot on the bottom-left is the cross-section of the negative of the LoG through the centre in any direction. Note that the zero crossings of the LoG occur at $x^2 + y^2 = 2\sigma^2$ which defines a circle of radius 2σ centred on the peak of the Gaussian function.

The bottom-right diagram shows a 5 × 5 kernel that approximates the shape if of the negative of the LoG function. This approximation is not unique. Its purpose is to capture the essential shape of the LoG function. It consists of a positive, central term surrounded by an adjacent, negative region whose values decrease as a function of distance from the origin, and a zero outer region. The coefficients must sum to zero so that the response of the kernel is zero in areas of constant intensity.

Filter kernels of arbitrary size (but fixed σ) can be generated by sampling the negative LoG function, and scaling the coefficients so that they sum to zero.

A more effective approach for generating a LoG kernel is sampling the function:

$$G(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}}$$

with appropriate σ, then convolving the resulting array with a Laplacian kernel, such as the kernel in slide 7.

Steps of the LoG algorithm

- The LoG algorithm includes these steps:

1. Convolving the LoG kernel with the image: $g(x,y) = [\nabla^2 G(x,y)] \star f(x,y)$.
2. Find the zero crossing of $g(x,y)$ to find the locations of edges in $f(x,y)$

- Since both the Laplacian and convolution operations are linear, we get:

$$g(x,y) = [\nabla^2 G(x,y)] \star f(x,y) = \nabla^2 [G(x,y) \star f(x,y)]$$

- This implies that we can achieve the same results by:

1. Smooth the image with a Gaussian filter using convolution.
2. Compute the Laplacian of the results.
3. Find the zero crossing of the output of the Laplacian.

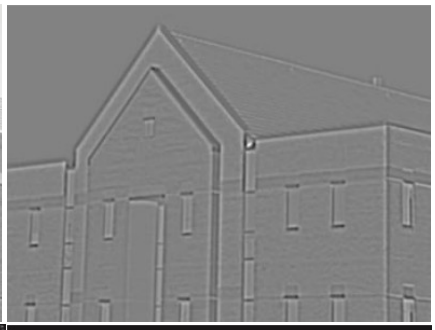
The application of LoG can be done in two different ways. However, due to the linear nature of both the Laplacian and the convolution operations, the preferred way is to:

1. Apply Gaussian smoothing filter by convoluting the image with a Gaussian kernel. This low pass filter the image and remove noise, but blurring edges (a bit).
2. Compute the Laplacian of this result where edges are highlighted.
3. Identify the edges by determining the zero crossing of the resulting image.

Matlab Image Processing Toolbox has a function ***edge*** which allows the LoG operation to be perform without you need to do the steps yourself.

Example of using LoG for edge detection

$f(x, y)$



LoG output
$g(x, y)$

Zero crossing threshold = 0

Zero crossing threshold = 4% of max intensity of *g*

Shown on top-left is a grayscale image of the front of a building.

After applying the Laplacian of a Gaussian, we get the top-right image which show the lines in the image in black lines (zero-crossing).

The bottom-left image is to determine zero-crossing using threshold of 0, but it generates lots of spurious features due to noise in image.

The bottom-right image is using a threshold of 4% of maximum value in $g(x, y)$ to obtain the zero-crossings. The noise is gone and edges are highlighted.

This is still not an ideal result. There is better and probably most used edge detection technique used for visual information processing known as the Canny edge detector. We will consider this next.

The Canny Edge Detector

- The Canny edge detector is based on three objectives:
  1. Low error rate: find all edges with no false and spurious results.
  2. Well localized edge points: location of edge points actually on edges.
  3. Single edge point response: return only one point for each true edge point.
- To achieve these objectives, Canny detector applies five steps:
  1. Apply Gaussian filter to smooth the image, thus removing noise.
  2. Find the intensity gradients of the filter image (i.e. 1st derivative), including both the **gradient magnitude** and **direction**.
  3. Apply **non-maximum suppression** to thin the edges.
  4. Apply **double threshold** to determine potential edges.
  5. Using **hysteresis method**, follow the strong edge points to produce the final definitive edge.

The Canny edge detector is one of the best and most popular edge detection algorithm. There are the five steps shown above in the slide.

Canny Detector – Step 2: Gradient Magnitude & Direction

- Step 1: The Filtering the image $f(x,y)$ with a Gaussian filter is similar to that of LoG edge detector. It removes noise from the image.
- Step 2: Compute the gradient at each pixel. Need to compute BOTH magnitude and direction:

$$M_s(x,y) = \sqrt{\left(\frac{\partial f_s(x)}{\partial x}\right)^2 + \left(\frac{\partial f_s(y)}{\partial y}\right)^2}$$

$$\alpha(x,y) = \tan^{-1}\left[\frac{\partial f_s(y)/\partial y}{\partial f_s(x)/\partial x}\right]$$

- Angle quantized to one of four directions: horizontal ($0°$), vertical ($90°$) and the two diagonals ($45°$, $135°$).

The first two steps are:

Step 1: Filtering the image $f(x,y)$ with a Gaussian filter is similar to that of LoG edge detector. It removes noise from the image. You can adjust the value of σ to match the required degree of filtering. High σ removes more noise, but will also blur edges more. The result is a smoothed image $f_s(x,y)$.

Using Matlab, the function ***fspecial*** generates the filter kernel for a Gaussian filter with the specified σ value.

Step 2: Compute the intensity gradients through out the image. This includes the first derivatives in x an y directions. These can be computed using the formula in slide 4 above, to obtain the magnitude $M_s(x,y)$ and the directional angle $\alpha(x,y)$, where

$$M_s(x,y) = \sqrt{\left(\frac{\partial f_s(x)}{\partial x}\right)^2 + \left(\frac{\partial f_s(y)}{\partial y}\right)^2}$$

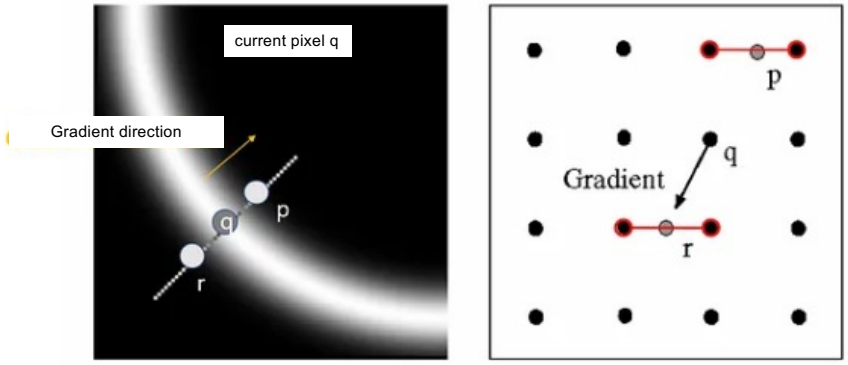
and

$$\alpha(x,y) = \tan^{-1}\left[\frac{\partial f_s(y)/\partial y}{\partial f_s(x)/\partial x}\right]$$

Note that the edge angle is then "quantized" to one of four directions: horizontal, vertical, and the two diagonals (i.e. $0°$, $90°$, $45°$, $135°$) respectively. In some algorithms, more directions than four are used.

Canny Detector – Step 3: Non-Maximum Suppression

- Step 3 of Canny is to use an edge thinning method to combat the smoothing effect of Gaussian smoothing.
  1. Compare intensity at q with neighbours along gradient direction p and r.
  2. Since q is maximum, set p and r to zero.
  3. Repeat for all pixels.



The problem of using Gaussian filter to reduce noise is that it tends to blur the image and makes the edge thicker. This has the undesirable effect of turning an edge into two edges, one on either side of the true edge.

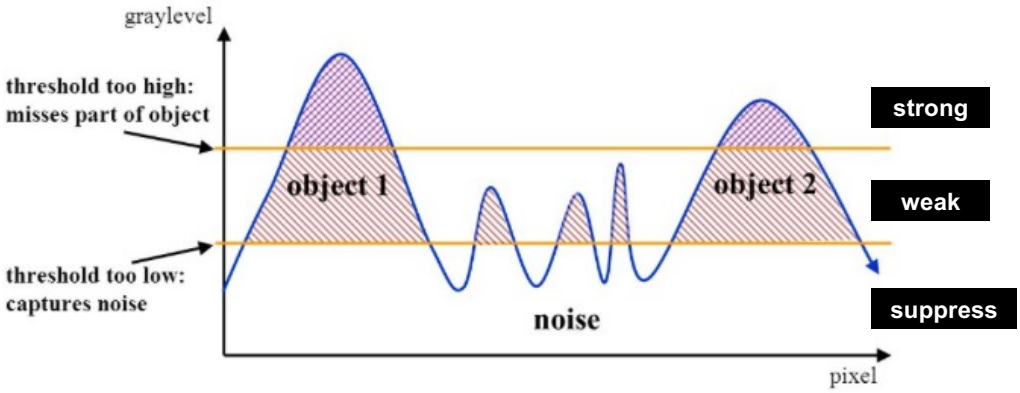
Step 3 of the Canny algorithm is to thin the blur edge by the following steps:

1. The gradient calculation from Step 2 identify a possible edge point q. We have both the gradient magnitude and the direction.. The direction perpendicular to the gradient is occupied by neighbours pixels p and r.
2. Compare p and r intensity with q. Since they are lower than q, set p and r intensity to zero.
3. Repeat steps 1 and two on all pixels detected along the edge.

In this way, all edges are thinned to single pixel.

Canny Detector – Steps 4: Double thresholding

- Noise can produce false edges even after Step 3.
- Use higher and lower thresholds to categorize each pixel.
- Gradient magnitude > high threshold → strong pixel.
- Low threshold ≤ Gradient magnitude ≤ high threshold → weak pixel.
- Gradient magnitude < low threshold → suppress pixel.

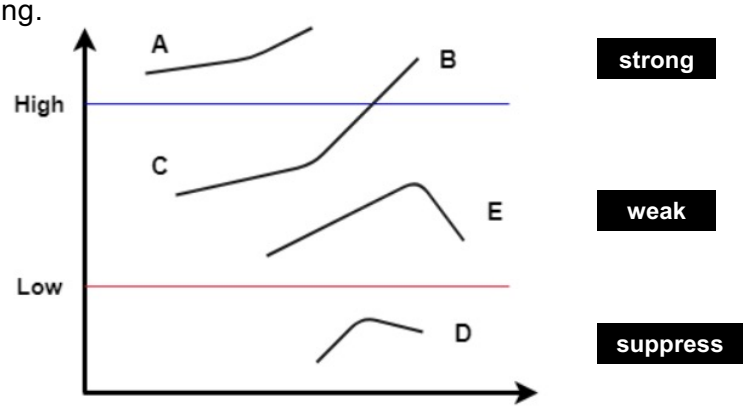


After step 3 of non-maximum suppression, the remaining edge pixels provide a more accurate and thinner representation of real edges in an image. However, there may still be false edges due to noise or colour variations. To account for these spurious edges, it is essential to remove edge pixels with a weak gradient value and preserve edge pixels with a high gradient value.

This is accomplished by selecting high and low threshold values, and classify every pixel gradient as one of three types: strong, weak and suppress. All suppressed pixels are NOT edge. All strong pixels are definitely on the edge. The weak pixels are the ”maybe” candidates, feeding into Step 5, edge tracking by hysteresis.

Canny Detector – Steps 5: Edge tracking by Hysteresis

- Finally, edge is tracked by its neighbourhood connections (hysteresis).
- A pixels are all strong. So A must be an edge.
- D pixels are all suppressed and therefore are not considered in Step 5.
- E pixels are all weak and none of their neighbours are strong – suppress.
- B is strong, but C is weak. However, C pixels are neighbour to strong, so reclassified as strong.



Step 5 involves "tracking" edges, but considering neighbourhood of each pixel. The diagram above shows five edge segments: A to E.

A and B are sure-edges as they are above 'High' threshold.

D is a sure non-edge.

Both 'E' and 'C' are weak edges but since 'C' is connected to 'B' which is a sure edge, 'C' is also considered as a strong edge.

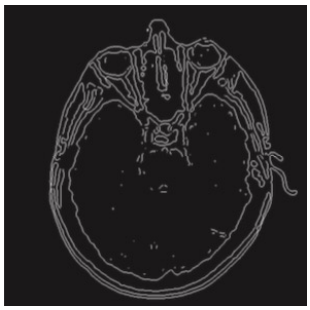
'E' is not connected to any strong pixels, therefore it is discarded.

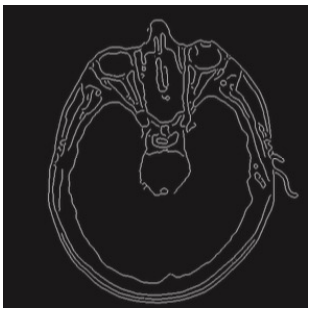
Compare Canny with other edge detection methods

$f(x, y)$

Edge detection with smoothing (5x5 square kernel) then thresholding

Edge detection with LoG method





Edge detection with Canny method

The top-left image shows a 512 × 512 head CT image. The goal is to extract the edges of the **outer contour** of the brain (the grey region in the image), the contour of the spinal region (shown directly behind the nose, toward the front of the brain), and the outer contour of the head. We wish to generate the thinnest, continuous contours possible, while eliminating edge details related to the grey content in the eyes and brain areas.

Top-right image shows the result of thresholding the gradient image that was first smoothed using a 5 × 5 averaging "square" kernel. The threshold required to achieve the result shown was 15% of the maximum value of the gradient image.

The bottom-left image shows the result obtained with the LoG edge-detection algorithm with a threshold of 0.002, σ = 3, and a kernel of size 19 × 19.

The best result is that of bottom-right, which was obtained using the Canny algorithm with TL = 0.05, TH = 0.15 (3 times the value of the low threshold), σ = 2, and a kernel of size 13 × 13.

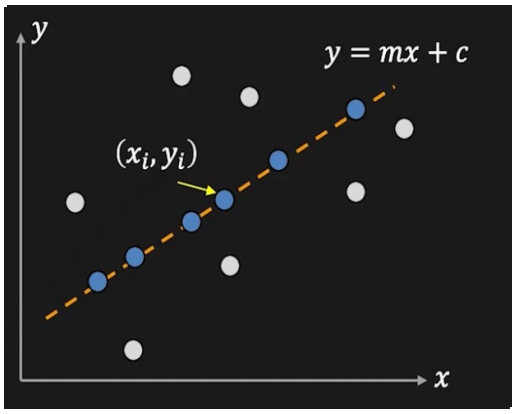
Comparison of Edge detection methods

Method	Pros	Cons
Sobel	Simple; detect edges and the orientations	Sensitive to noise; inaccurate
Laplacian + zero crossing	Detect edges and direction; isotropic	Sensitive to noise; interaction between nearby edges
Laplacian of Gaussian (LoG)	Correct places of edges; handle different areas and scales	Malfunction at curves and corners; cannot find orientations
Canny	Low error rate; good localization; accurate; not sensitive to noise	More complex; sometimes produce false zero crossings

This table summarizes the various edge detection techniques discussed earlier. Without doubt, the Canny method produces best results, but computationally more complex. Canny method is suitable as a programmed solution while the others are more suitable for direct implementation on digital hardware (i.e. custom chip or FPGA implementations).

The Hough Transform – Basic Idea

- Previous method detected edge points (x_i, y_i) as shown here.
- How to detect line $y = mx + c$?



- Consider the point (x_i, y_i). Its equation is given by:

$$y_i = mx_i + c$$

Parameter space

$$c = -mx_i + y_i$$

The Hough transform is a technique which can be used to isolate features of a particular shape within an image. Because it requires that the **desired features be specified** in some parametric form, the Hough transform is most commonly used for the detection of regular curves such as lines, circles, ellipses, etc. or any feature boundaries which can be described by regular curves.

The main advantage of the Hough transform technique is that it is tolerant of gaps in feature boundary descriptions and is relatively unaffected by image noise.

To explain the idea of Hough Transform which was introduced in 1960's, consider a number of edge points detected by previous methods. We want to find a line in all these edge points.

By inspection, you might consider the point in blue belong to a line with equation:

$$y = mx + c\ .$$

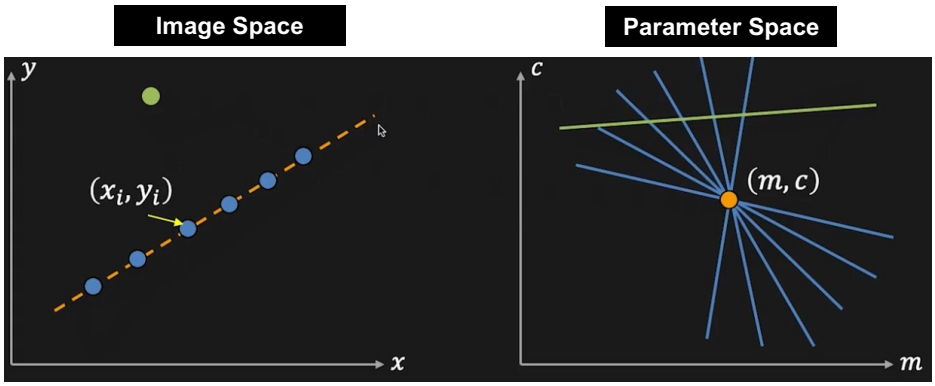
Let us know consider one specific point on this line, say (x_i, y_i). We substitute this point to the line equation. Now we have the equation on the right:

$$c = -mx_i + y_i$$

Since (x_i, y_i) are given and fixed, this equation describes all lines with (m, c) values that pass through the point (x_i, y_i).

In other words, (x, y) coordinate is in image space, and(m, c) values are the parameters of lines, but now in parameter space.

The Hough Transform – Image Space vs Parameter Space



$$y_i = mx_i + c \quad \Longleftrightarrow \quad c = -mx_i + y_i$$

- All lines through edge point (x_i, y_i) maps onto the blue line in the parameter space.
- Another point on the line in image space maps to another line in the parameter space but intersect at the same (m, c) values.
- Now map a few more points on the edge line, will result in same intersection.

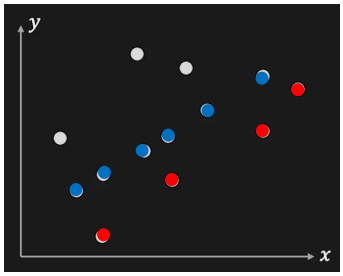
For a given point (x_i, y_i), all lines through this point in the image space maps to a single line in the parameter space.

So each blue point on the left maps to a blue line on the right. The intersection of these blue lines provides the parameter (m, c) for the line in the image space. Therefore, the equation of the line is identified. That is, line is detected.

Consider the GREEN edge point. This will also result in the green line in the parameter space, but it does not provide a common intersection to all the other blue lines (i.e. it intersect them at difference locations).

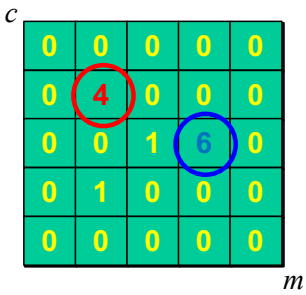
The Hough Transform – Line Detection Algorithm

- Step 1: Quantize parameter space (m, c).
- Step 2: Create a counting array $H(m, c)$.
- Step 3: Set $H(m, c) = 0$ for all (m, c).
- Step 4: For each **edge point** (x_i, y_i)
  $H(m, c) = H(m, c) + 1$
  So for all points on the straight line, this increase the count at (m, c).
- Step 5: Identify the local maxima in $H(m, c)$.



$f(x, y)$

image



$H(m, c)$

Parameter counting bins

Now we have defined the Hough Transform as one that maps edge points in the image space to lines in the parameter space. Provided that we find the common intersection, we can detect the line. How is this achieved in a step-by-step algorithm?

The above slides explain how this can be achieved.

Step 1: Divide the parameter space (m, c) into discrete values.

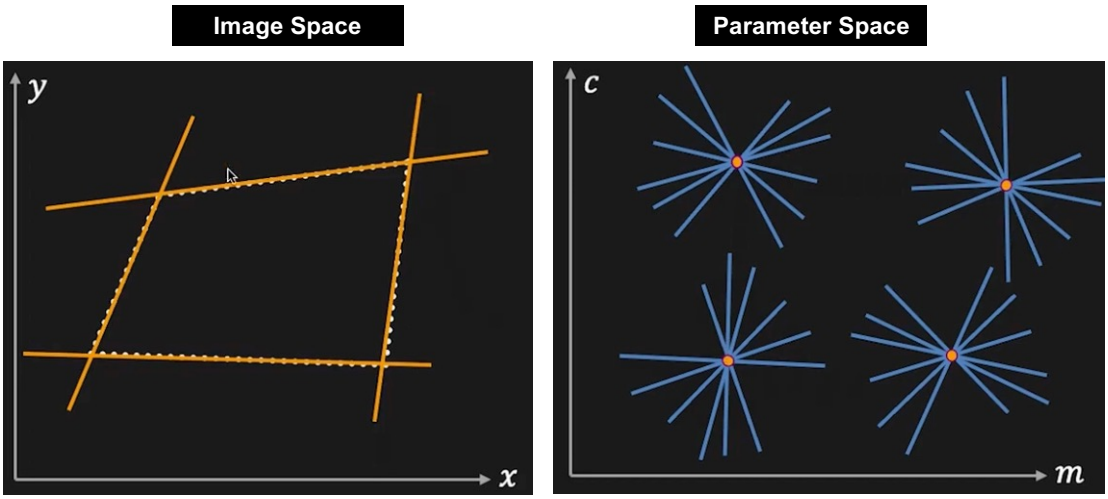
Step 2: Create a new array $H(m, c)$ for each discrete parameter values. Each cell in the array stores the number of times that parameter pair is detected.

Step 3: Initialize the array to have zero.

Step 4: For each edge point, calculate the (m, c) values and increment the corresponding array bin count by 1.

Step 5: After this is done for ALL edge points, the bins with the largest count within a locality identify the (m, c) values of the detected line.

The Hough Transform – Multiple line detection



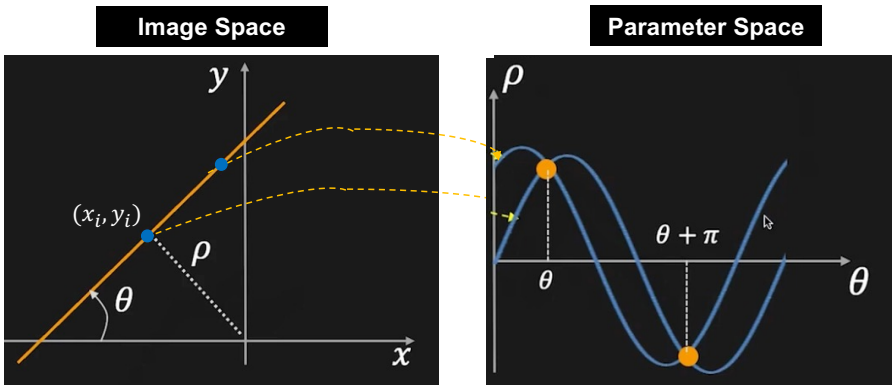
Here is an example of using Hough Transform on an image with four lines.

This results in the parameter space sets of lines that identifies four significant intersection with high frequency of occurrence. These provide the four set of parameters for the four lines in the original image.

The Hough Transform – Better Parameters

- Problem with using (m, c) parameter space.
- Range of slope of line is huge:
  $$-\infty \leq m \leq \infty$$
- Not viable for limited size of counting array.

- **Solution**: do not map line to $y = mx + c$
- Instead use a different equation for a line:
  $$x \sin\theta + y \cos\theta + \rho = 0$$
- The orientation θ is finite: $0 \leq \theta \leq \pi$
- The distance ρ from origin is also finite.



$x \sin\theta + y \cos\theta + \rho = 0$ $\qquad$ $x \sin\theta + y \cos\theta + \rho = 0$

The previous discussion works in theory, but not in practice. The reason is that the parameter m, the gradient, has the range of ±infinity! This means that the parameter space bin array is infinite, or at least very large.

The solution is NOT to use the line equation with (m, c), but use another equation to represent the straight line.

For the point (x_i, y_i) shown above, which is a distant ρ from the origin and is on a line which sustain an angle θ relative the x-axis. Simple trigonometry shows that:

$$x_i = \rho \sin\theta, \text{ and } y_i = \rho \cos\theta$$

Hence, $\quad x_i \sin\theta = \rho \sin^2\theta$, and $y_i \cos\theta = \rho \cos^2\theta$.

Summing these two equation gives us:

$$x_i \sin\theta + y_i \cos\theta = \rho \sin^2\theta + \rho \cos^2\theta = \rho.$$

Therefore, if we use the parameter space (θ, ρ) instead of (m, c), all straight lines in the image space through (x_i, y_i) now maps to a sinewave in the parameter space.

Two points on the line that is at an angle θ_j to the x-axis and it perpendicular distant form the original is ρ_j, will map to two sinewaves that intersection at (θ_j, ρ_j).

The reasons that this mapping is much better is because both θ is always between 0 and π. The perpendicular distance ρ, is also finite.

The Hough Transform – Design consideration

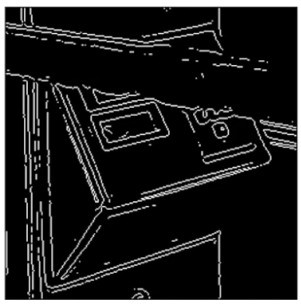
- What is the dimension of the parameter counting array?
  - Too many bins, noise will cause lines to be missed.
  - Too few bins, different lines will merge together.
- How many lines?
  - Count the peaks in the array (thresholding),
- How to handle inaccurate edge locations?
  - Increment nearby bins instead of just individual bins.

In using Hough Transform, there are a few practical decisions to be made. How should one quantize the parameter space. For example, we know the angle θ is between 0 and π. How many incremental angle steps should we use? This decision is image dependent and require engineering judgement depending on what we want to achieve.

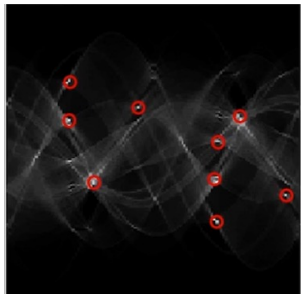
Also, once we have the accumulated counts in the parameter space, there will be many high values. How should the peaks be counted and in how large a local area?

Example of Hough Transform in line detection

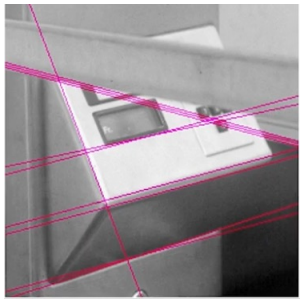
Input image



Detected edge



Hough Transform $H(\theta, \rho)$



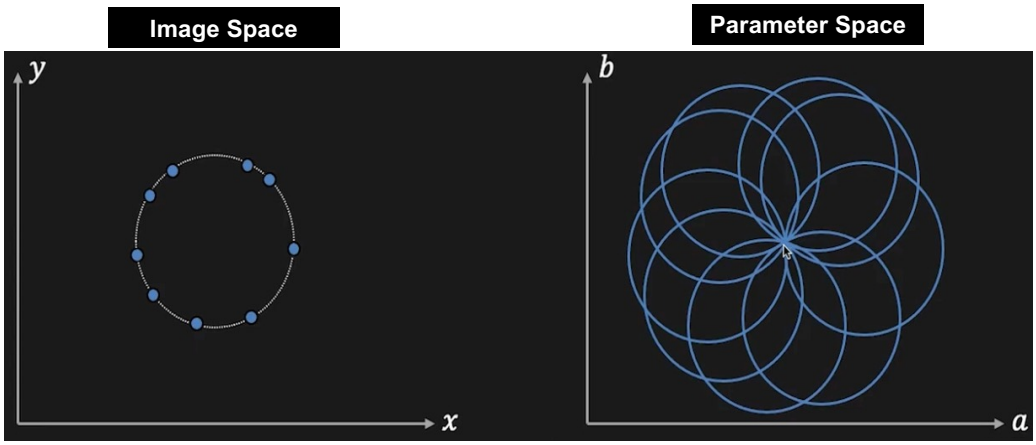
Detected lines

Here is an example of using Hough Transform to identify all the straight lines in this image. The top-right image shows all the edge points after edge detection.

The bottom-left image is the Hough Transform in the parameter space showing the frequency count of the parameters for the edge points. The high count bins are highlighted in red. There are nine lines identified.

The bottom-right image show all the identified lines.

Hough Transform: Detection of Circle (known r)



$$(x_i - a)^2 + (y_i - b)^2 = r^2 \quad \Longleftrightarrow \quad (a - x_i)^2 + (b - y_i)^2 = r^2$$

- For circle of known radius, and a give point (x_i, y_i).
- All circles through this point maps to a circle in the parameter space.
- The intersection of all edge points gives the parameter (a, b).

The Hough transform can also be used to detect circuits from the edge points image. For simplicity, let us assume that we KNOW the radius of the circuit. Then the parameter space is in (a, b), the centre of the circle.

The mapping from image space to parameter space is the equation shown above, which is also an equation for a circle. That is, given a point (x_i, y_i) in the image space, all circles through this point also maps to a circuit in the parameter space.

Therefore the parameter (i.e. the centre coordinate of the circle) is given by the common intersections of all the circles for the blue points in the image space as shown.

Hough Transform: Circle detection example

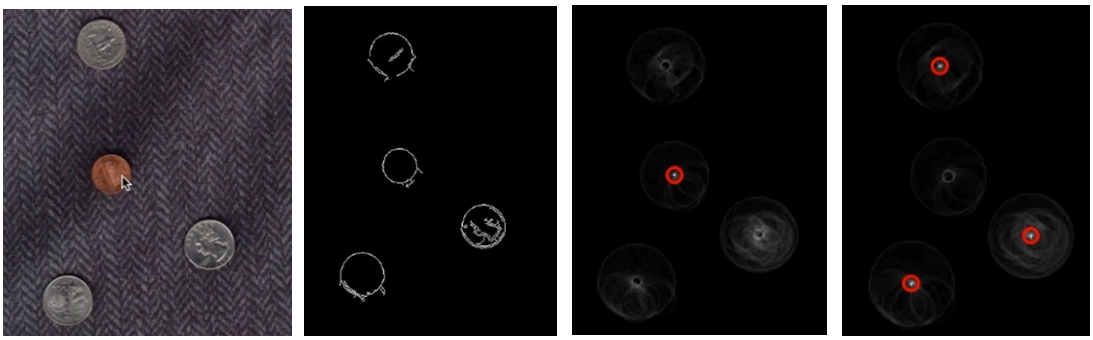


Image of coins | Edge points | Hough Transform $H_1(a,b)$ for penny ($r = r_1$) | Hough Transform $H_2(a,b)$ for quarters ($r = r_2$)

Finally, here is an example of using Hough Transform to identify the location of two types of coins: penny with radius r_1 and quarters with radius r_2.

After the edge points are found, we use two Hough Transforms, one for pennies and another for quarters. They result in the clearly identified dots in the two separate parameter spaces as shown.