

# Boundary Detection

Shree K. Nayar

Monograph: FPCV-2-2

Module: Features

Series: First Principles of Computer Vision

Computer Science, Columbia University

June 10, 2022

[FPCV Channel](#)

[FPCV Website](#)

## Boundary Detection

Shree K. Nayar

Columbia University

Topic: Boundary Detection, Module: Features  
First Principles of Computer Vision

1

## Boundary Detection

We need to find Object Boundaries from Edge Pixels.

Topics:

- (1) Fitting Lines and Curves to Edges
- (2) Active Contours (Snakes)
- (3) The Hough Transform
- (4) Generalized Hough Transform

2

Now that we know how to find the edges in an image, we want to use these edges to produce clean outlines of the objects in the image. This is the problem of boundary detection. Simply put, we want to go from edge pixels to continuous object boundaries.

First, we will discuss fitting lines and curves to edges. Given a set of edges, the task is to fit a low-order polynomial to the edges. We will see how this can be set up as a system of linear equations that can be solved efficiently. Next, we will talk about active contours, also known as snakes. These are widely used in computer vision, in particular in fields such as medical imaging. Imagine that we are interested in finding a specific object in the image. The image, of course, has a variety of other things inside of it. To guide our detection, we start by sketching a rough outline (contour) of the object. The algorithm then takes this contour and iteratively modifies it until it latches on, like a rubber band, to the actual boundary of the object. We refer to this type of an iteratively modified boundary as an active contour.

In boundary detection, a challenging problem is figuring out which edges belong to the boundary that we are looking for. This is a type of “inlier-outlier” problem. A technique called the Hough transform— invented in the 1960s — gives us an elegant way of solving this problem. We will describe the Hough transform and show how it can be used to find simple shapes, such as lines and circles, that can be described using equations with a small number of parameters. The Hough transform can detect such shapes robustly even when the image is complex.

We will end with a generalization of the Hough transform. Consider the case where we are not given the parametric equation for the shape of the object we are looking for; an example would be a hand-drawn sketch. It turns out that we can generalize the Hough transform to apply it to such complex shapes as well. We will see how the generalized Hough transform can be implemented and what its limitations are.

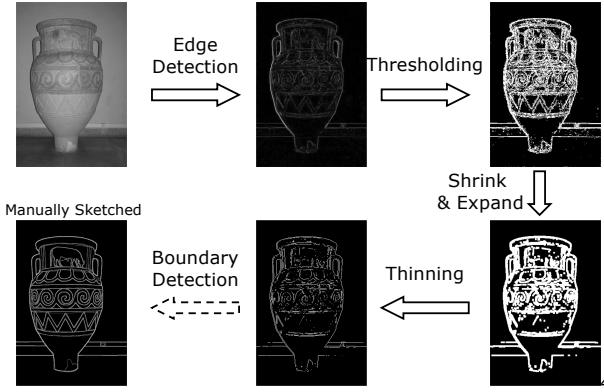
## Fitting Lines and Curves

Shree K. Nayar

Columbia University

Topic: Boundary Detection, Module: Features  
First Principles of Computer Vision

### Preprocessing Edge Images



3

4

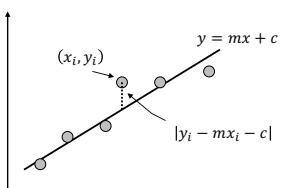
Imagine we are given a set of edges in an image and we want to fit either a straight line or a curve to the edges. Shown here on the top left is an image of a vase for which we want to produce a boundary detection output that looks like the image in the bottom left corner. This desired output has been manually sketched by an artist, and hence is very clean with clear boundaries. It is worth stating upfront that such a high-quality output is only aspirational and would be unrealistic to expect from any algorithm. First, we will apply edge detection to the input image, yielding an edge map with the strength, or magnitude, of the edge at each pixel. We can then threshold this edge map to produce a binary edge image. This image can be processed using the techniques we discussed in binary image processing. For instance, if we shrink and expand the edges, we end up with the image in the bottom right corner. Shrinking removes all the isolated edges in the image, after which the remaining edges are expanded again. Since these edges will probably be a little thicker than we would like, we can apply a thinning algorithm. Now the challenge is to go from this edge map to boundaries.

Let us consider the simple problem of fitting a line to the set of edges shown here. In other words, we want to find the slope  $m$  and the intercept  $c$  corresponding to the line that best represents the set of edges. We will start by setting up an energy, or cost, function. We can define the energy  $E$  as the average of the square of the vertical distances between each point (edge) and the line that we are trying to estimate. The vertical distance for the point  $(x_i, y_i)$  is simply  $y_i - mx_i - c$ . We then square this distance for each of the points, sum all of the distances, and then divide by the number of points,  $N$ , to obtain the average  $E$ . To find the “best” line for these edges, we need to find the  $m$  and  $c$  that minimize  $E$ . We know how to do that — we find the partial derivatives of  $E$  with respect to  $m$  and  $c$

### Fitting Lines to Edges

Given: Edge Points  $(x_i, y_i)$

Task: Find  $(m, c)$



Minimize: Average Squared Vertical Distance

$$E = \frac{1}{N} \sum_i (y_i - mx_i - c)^2$$

Least Squares Solution:

$$\frac{\partial E}{\partial m} = \frac{-2}{N} \sum_i x_i (y_i - mx_i - c) = 0 \quad \frac{\partial E}{\partial c} = \frac{-2}{N} \sum_i (y_i - mx_i - c) = 0$$

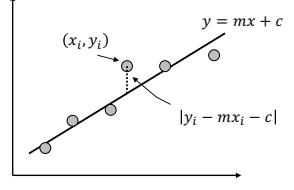
5

and set them equal to 0. Remember that when we find the derivative, we can take the derivative inside the summation because both are linear operators.

From the two partial derivatives, we get these closed-form expressions for  $m$  and  $c$ . We have therefore found a line that fits the edges.

### Fitting Lines to Edges

Given: Edge Points  $(x_i, y_i)$   
 Task: Find  $(m, c)$



Solution:

$$m = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2} \quad c = \bar{y} - m\bar{x}$$

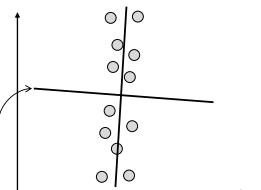
where:  $\bar{x} = \frac{1}{N} \sum_i x_i \quad \bar{y} = \frac{1}{N} \sum_i y_i$

6

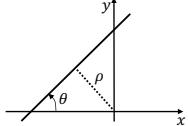
The above solution, unfortunately, does not always work well. Imagine using it to fit a line to the set of points shown here. We can see that the line we are looking for — the line that best represents these points — is a more or less vertical line that passes through the set of points. It turns out, however, that the line that we end up computing is actually the horizontal line shown here. It is virtually perpendicular to the line that we are looking for! This happened because we were minimizing an energy that represents the vertical distance of each point from the line. Note that the vertical distance from any of these points to the desired vertical line is fairly large. Thus, when we set up a fitting problem such as this, we need to be careful about how we formulate the energy function. A more reasonable thing to do is to minimize the average of the perpendicular distances between the points and the line. For that, it is convenient to use a different parameterization of the straight line —  $x \sin \theta - y \cos \theta + \rho = 0$  — which we discussed in the lecture on binary images. Here,  $\theta$  is the angle that the line makes with respect to the horizontal axis, and  $\rho$  is the shortest distance from the line to the origin.

### Fitting Lines to Edges

Problem: When the points represent a vertical line.  
 Line that minimizes E!



Solution: Use a different line equation



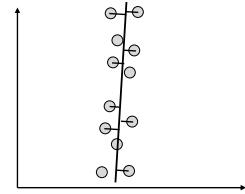
$$x \sin \theta - y \cos \theta + \rho = 0$$

7

Note that the expression  $x_i \sin \theta - y_i \cos \theta + \rho$  is the perpendicular distance of the point  $(x_i, y_i)$  from the line. This equation should look familiar since it is the same as the one we used to find the axis of minimum second moment of an object in a binary image. We now know that if we are given a set of points, we can treat them like points on a binary object and simply find the axis of minimum second moment. That axis will be the line that minimizes the average of the square of the perpendicular distances of the points from the line.

### Fitting Lines to Edges

Problem: When the points represent a vertical line.



Minimize: Average Squared Perpendicular Distance

$$E = \frac{1}{N} \sum_i \underbrace{(x_i \sin \theta - y_i \cos \theta + \rho)^2}_{\text{Perpendicular Distance}}$$

(See Binary Img. Processing Lecture)

8

Now, let us look at the problem of fitting a curve to a set of points. Let us say the curve we wish to fit can be described as a third-order polynomial, although it can be of any order. We want to find the parameters of the polynomial that best fits these points. For simplicity, let us return to our earlier distance metric — the vertical distance. Then, we can use the energy function shown here, which is the average of the squared vertical distance between each point and the polynomial that we are fitting. To minimize this function, we take its derivatives with respect to each of the unknown parameters ( $a, b, c$  and  $d$ ) and set them equal to 0 to get a system of equations. We can then solve this system of equations to find the parameters  $a, b, c$  and  $d$  of the polynomial.

### Fitting Curves to Edges

Given: Edge Points  $(x_i, y_i)$

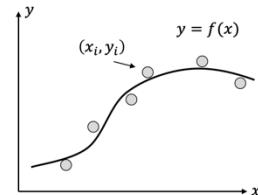
Task: Find polynomial

$$y = f(x) = ax^3 + bx^2 + cx + d$$

that best fits the points

Minimize:

$$E = \frac{1}{N} \sum_i (y_i - ax_i^3 - bx_i^2 - cx_i - d)^2$$



Solve the Linear System Using Least Squares Fit by:

$$\frac{\partial E}{\partial a} = 0 \quad \frac{\partial E}{\partial b} = 0 \quad \frac{\partial E}{\partial c} = 0 \quad \frac{\partial E}{\partial d} = 0$$

Closed-form solution cumbersome when many unknowns

9

The problem here is that, given any polynomial, we will have to find the derivatives to create the system of equations, and then solve the system to find the parameters of the polynomial. That is quite cumbersome to do on a case-by-case basis. Is there a more general technique that we can use instead?

An alternative approach is to construct a system of linear equations as follows. Let us assume we wish to fit a third-order polynomial. By simply plugging each point  $(x_i, y_i)$  into the polynomial we get an equation. If we have a total of  $n$  points, we get the system of  $n$  equations shown here. We wish to find the four unknowns —  $a$ ,  $b$ ,  $c$  and  $d$  — using these equations. If the number of points is larger than the number of unknowns, which is typically the case, then we have an over-determined linear system of equations.

### Fitting Curves to Edges

Solving as a Linear System:

$$y_0 = ax_0^3 + bx_0^2 + cx_0 + d$$

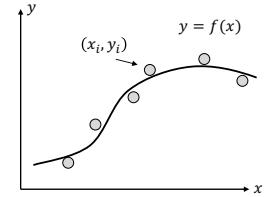
$$y_1 = ax_1^3 + bx_1^2 + cx_1 + d$$

⋮

$$y_i = ax_i^3 + bx_i^2 + cx_i + d$$

⋮

$$y_n = ax_n^3 + bx_n^2 + cx_n + d$$



Given many  $(x_i, y_i)$ 's, this is an over-determined linear system with four unknowns  $(a, b, c, d)$ .

10

Let us take a look at how we can solve such an over-determined linear system of equations in a more general setting, where we have  $m$  unknowns and  $n$  observations (points). Here, the observations are denoted as  $(x_{ij}, y_i)$ . Note that  $x_{ij}$  has two subscripts where  $i$  corresponds to the index of the data point itself — in our case, the edge — and  $j$  represents the power that  $x_i$  is raised to in the polynomial equation (see previous slide). We can rewrite this system in matrix form as  $X\mathbf{a} = \mathbf{y}$ , where the matrix  $X_{n \times m}$  and the vector  $\mathbf{y}_{n \times 1}$  are known and we wish to find the coefficient vector  $\mathbf{a}_{m \times 1}$ . Our first instinct may be to find the inverse of  $X$ , but  $X$  is not a square matrix and is therefore not invertible. We can multiply both sides of the equation with the transpose of  $X$ , since we know that  $X^T X$  is an  $m \times m$  matrix — a square matrix. We can therefore find the inverse of  $X^T X$  to solve for  $\mathbf{a}$ . This approach to solving an over-determined system of equations is called the pseudo-inverse method, where the pseudo inverse is  $X^+ = (X^T X)^{-1} X^T$ . This is a general technique for solving any system of over-determined linear equations.

### Solving a Linear System

An over-determined linear system with  $m$  unknowns  $\{a_j\}$  ( $j = 0, \dots, m$ ) and  $n$  observations  $\{(x_{ij}, y_i)\}$  ( $i = 0, \dots, n$ ) ( $n > m$ ) can be written in a matrix form.

$$\left[ \begin{array}{cccc} x_{00} & x_{01} & \dots & x_{0m} \\ x_{10} & x_{11} & \dots & x_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n0} & x_{n1} & \dots & x_{nm} \end{array} \right] \left[ \begin{array}{c} a_0 \\ a_1 \\ \vdots \\ a_m \end{array} \right] = \left[ \begin{array}{c} y_0 \\ y_1 \\ \vdots \\ y_n \end{array} \right]$$

$X_{n \times m}$  Known  $\mathbf{a}_{m \times 1}$  Unknown  $\mathbf{y}_{n \times 1}$  Known

$X\mathbf{a} = \mathbf{y}$   
 $X_{n \times m}$  is not a square matrix and hence not invertible.

Least Squares Solution:

$$X^T X \mathbf{a} = X^T \mathbf{y} \Rightarrow \mathbf{a} = (X^T X)^{-1} X^T \mathbf{y} \quad X^+ = (X^T X)^{-1} X^T$$

(Pseudo Inverse)

MATH PRIMER

11

## Active Contours

Shree K. Nayar

Columbia University

Topic: Boundary Detection, Module: Features  
First Principles of Computer Vision

12

### What is an Active Contour?

Given: Approximate boundary (contour) around the object



Image

Task: Evolve (move) the contour to fit exact object boundary

Active Contour:

Iteratively “deform” the initial contour so that:

- It is near pixels with high gradient (edges)
- It is smooth

Also called Snakes

[Kass 1987]

13

Next, let us talk about the concept of active contours, also referred to as snakes. This is a powerful tool for finding the boundaries of objects in an image. Shown here is an image of a coin with a rough initial contour (dotted curve) drawn around it. This contour could even be sketched by hand. We want this initial contour to evolve over time so that it finally latches on to the boundary of the coin itself. In other words, an active contour iteratively deforms an initial contour so that it ends up at the pixels that lie on the boundary of the object. While implementing this technique, we want to make sure that the final contour does not have knots and twists in it; it should be smooth. Active contours are popular in many domains, in particular in the field of medical imaging.

One of the powerful features of active contours is that they can be used to track objects over a video sequence. For instance, in the case of the sequence of the moving lips shown here, an active contour is first used to find the outline of the lips in the first frame of the sequence. The contour for this frame can be used as the initial contour for the next frame. Therefore, we can use an active contour to effectively track the boundary of an object even as it deforms over time. Similarly, if we are given the video of an object that is taken while moving around the object, such as the car shown here, we can use an active contour to robustly find the boundary of the object even as its perspective changes with time. These examples demonstrate why active contours are powerful and have wide ranging applications.

### Power of Deformable Contours

Boundaries could deform over time



Boundaries could deform with viewpoint



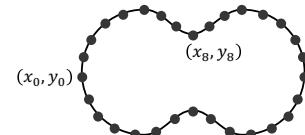
Boundary Tracking: Use the boundary from the current image as initial boundary for the next image.

14

We represent the contour as a set of  $n$  points, called the control points. The sampling of the contour is uniform, such that the distance between consecutive control points is fixed. We can therefore view the contour as a set of straight line segments of equal length linked together.

### Representing a Contour

Contour  $\mathbf{v}$ : An ordered list of 2D vertices (control points) connected by straight lines of fixed length



$$\mathbf{v} = \{v_i = (x_i, y_i) \mid i = 0, 1, 2, \dots, n - 1\}$$

15

Let us consider the simple case of finding the boundary of the coin shown here using an active contour. Shown as the blue dotted curve is the initial contour that is sketched by the user. We need to apply some forces to this initial contour to draw it closer to the coin. Since the boundary of any object is likely to have high gradients, it would make sense to use the gradient of the image to generate the forces that act on the contour. The image in the center shows the square of the magnitude of the gradient of the image. Now the question is, how do we attract the contour closer to the object when the gradients are sitting so far away? The trick is to blur the gradients so that they can influence contour points that are distant. The image on the right shows the square of the magnitude of the gradient after the gradient has been smoothed using a Gaussian function with standard deviation  $\sigma$ . This image can be viewed as a force field, or a potential field, that acts on the contour. Ultimately, our goal is to arrive at a contour shape and location for which the sum of the square of the gradient magnitudes computed over all the control points of the contour is maximum. Maximizing this is equivalent to minimizing the negative of the summation in [1]. This is referred to as the image term  $E_{image}$ . The reason we pose this as a minimization and not a maximization problem is that later we plan to incorporate additional terms that we want to minimize.

### Attracting Contours to Edges



$$\begin{aligned} & \text{Maximize Sum of Gradient Magnitude Square} \\ & \equiv \text{Minimize } -\text{ve (Sum of Gradient Magnitude Square)} \\ & \equiv \text{Minimize } E_{image} = -\sum_{i=0}^{n-1} \|\nabla n_\sigma * I(v_i)\|^2 \quad [1] \end{aligned}$$

16

### Contour Deformation: Greedy Algorithm

1. For each contour point  $v_i$  ( $i = 0, \dots, n - 1$ ), move  $v_i$  to a position within a window  $W$  where the energy function  $E_{image}$  for the contour is minimum.

2. If the sum of motions of all the contour points is less than a threshold, stop. Else go to Step 1.



Greedy solution might be suboptimal and slow.

17

### Sensitivity to Noise and Initialization



Contour fitted to gradient magnitude

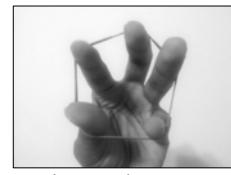
18

Our first algorithm for minimizing  $E_{image}$  is based on the greedy approach. We are going to move each contour point,  $v_i$ , within a small window around it to the position for which  $E_{image}$  is a minimum. Since it is a greedy approach, we do this in a sequential manner, one control point at a time. We apply the algorithm iteratively until the change in  $E_{image}$  with respect to the previous iteration is less than a threshold. This simple algorithm works reasonably well, but since it is greedy, it is not guaranteed to yield the optimal solution. This is especially a problem when the image is noisy, in which case, the gradient image and the force field it produces is also noisy. The slide on the right shows the final result of applying the algorithm; the contour does latch on to the boundary of the coin but includes some knots.

We would like to add more constraints to make the contour behave like a physical object with certain properties.

One of those properties is elasticity, where the contour behaves like a rubber band that contracts. We also want it to be smooth, like a metal strip, so it is not prone to sudden twists and turns. If we think about the contour as a physical object made of some material, imposing the above constraints — elasticity and smoothness — is equivalent to minimizing the internal bending energy of the object. We will refer to the internal bending energy as  $E_{contour}$ , which is a weighted sum of the elastic term  $E_{elastic}$  and the smoothness term  $E_{smooth}$ . Next, we will derive these two terms so they can be added to our original image term  $E_{image}$ . The weights  $\alpha$  and  $\beta$  can be adjusted based on the application to trade off elasticity for smoothness and vice versa.

### Making Contours Elastic and Smooth



Elastic and contracts like a rubber band



Smooth like a metal strip

Minimize Internal Bending Energy of the Contour:

$$E_{contour} = \alpha E_{elastic} + \beta E_{smooth}$$

$(\alpha, \beta)$ : Control the influence of elasticity and smoothness

19

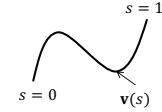
Now let us see how we can formulate elasticity and smoothness. Imagine that we have a contour like the one shown on the right. We will eventually discretize it, but for now we will consider it to be continuous. The contour is denoted as  $\mathbf{v}(s)$  with two coordinates  $x(s)$  and  $y(s)$ , where  $s$  is the parameter that represents the length along the contour and it goes from zero to one. Note that to minimize elasticity we want to minimize the rate of change of the contour and to maximize smoothness we want to minimize the curvature of the contour. Therefore, for the elastic term we use is the square of the first derivative of  $\mathbf{v}$  and for the smoothness terms we use the square of the second derivative of  $\mathbf{v}$ . For a discrete contour with control points  $\mathbf{v}_i$ , the above derivatives can be computed using the finite difference equations shown at the bottom.

### Elasticity and Smoothness

For point  $0 \leq s \leq 1$  on continuous contour  $\mathbf{v}(s) = (x(s), y(s))$ :

$$E_{elastic} = \left\| \frac{d\mathbf{v}}{ds} \right\|^2$$

$$E_{smooth} = \left\| \frac{d^2\mathbf{v}}{ds^2} \right\|^2$$



Discrete approximations at control point  $\mathbf{v}_i$ :

$$E_{elastic}(\mathbf{v}_i) = \left\| \frac{d\mathbf{v}}{ds} \right\|^2 \approx \|\mathbf{v}_{i+1} - \mathbf{v}_i\|^2 = (x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2$$

$$\begin{aligned} E_{smooth}(\mathbf{v}_i) &= \left\| \frac{d^2\mathbf{v}}{ds^2} \right\|^2 \approx \|(\mathbf{v}_{i+1} - \mathbf{v}_i) - (\mathbf{v}_i - \mathbf{v}_{i-1})\|^2 \\ &= (x_{i+1} - 2x_i + x_{i-1})^2 + (y_{i+1} - 2y_i + y_{i-1})^2 \end{aligned}$$

20

### Elasticity and Smoothness

Internal bending energy along the entire contour:

$$E_{contour} = \alpha E_{elastic} + \beta E_{smooth}$$

where:

$$E_{elastic} = \sum_{i=0}^{n-1} [(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2]$$

$$E_{smooth} = \sum_{i=0}^{n-1} [(x_{i+1} - 2x_i + x_{i-1})^2 + (y_{i+1} - 2y_i + y_{i-1})^2]$$

21

### Combining the Forces

Image Energy,  $E_{image}$ : Measure of how well the contour latches on to edges

Internal Energy,  $E_{contour}$ : Measure of elasticity and smoothness

Total Energy of Active Contour:

$$E_{total} = E_{image} + E_{contour}$$

Minimize the Total Energy

22

Here are the elastic and smoothness terms computed over the entire contour. These are combined using the weights alpha and beta to get the internal bending energy, which we refer to as the contour term,  $E_{contour}$ . The contour term is added to the image term  $E_{image}$  to get the total energy  $E_{total}$ . It is this total energy that we wish to minimize.

To minimize the total energy  $E_{total}$ , we will once again use a greedy approach. We first uniformly sample the contour to get  $n$  contour points. In order to ensure that the algorithm works well, we need to uniformly sample the contour after each iteration of the algorithm. In each iteration, for each control point, we find the point within a small window for which the total energy is a minimum, and move the control point to that point. We repeat this for all the control points to obtain the new contour. If we find that the sum of the displacements of all the control points with respect to the previous iteration is small, we terminate the algorithm. Else, we return to step 1 and run another iteration of the algorithm.

### Contour Deformation: Greedy Algorithm

1. Uniformly sample the contour to get  $n$  contour points.
2. For each contour point  $v_i$  ( $i = 0, \dots, n - 1$ ), move  $v_i$  to a position within a window  $W$  where the energy function  $E_{total}$  for the entire contour is minimum.

$$E_{total} = E_{image} + E_{contour}$$

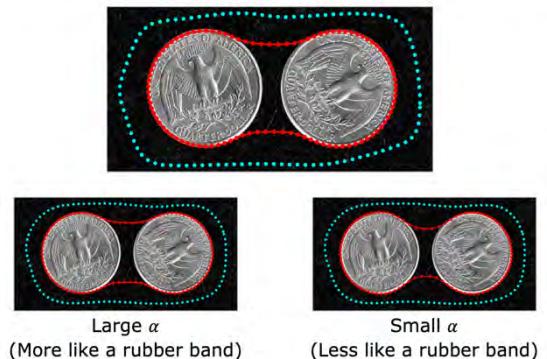
3. If the sum of motions of all the contour points is less than a threshold, stop. Else go to Step 1.



23

Shown here is the performance of the above algorithm. In this case, the image includes two coins and noise in the background. In the bottom row are shown two results, one with a large value for alpha and the other for a small value. When alpha is large, as expected, the elastic term has a stronger influence and the contour behaves like a stiffer rubber band. When alpha is low, the image term dominates and draws the contour closer to the edges of the coins in the region between the two coins.

### Result: Boundary Around Two Objects



25

Now let us discuss a few issues related to active contours. Firstly, we can modify the above formulation in many different ways. For example, we could add an extra term that penalizes the deviation of the contour from a prior model of the shape of the object. In this case, the final result we end up with will not only seek to satisfy all the previous constraints—the image, elastic and smoothness constraints — but also try to make the final contour similar in shape to the prior shape model.

### Active Contours: Comments

- Additional energy constraints can be added
  - Penalize deviation from prior model of shape
- Requires good initialization
  - Edges cannot attract contours that are far away
- Elasticity makes contour contract
  - Replace contracting force with ballooning force to expand

26

One issue with contours is that they require good initialization. If our starting contour is really far away from the real object, it is going to be difficult to create a force field that is able to draw it to the object. In this case, it is likely that it will instead latch on to other objects in the image. Finally, in our current formulation, the contour energy term, in effect, applies forces on the contour that make it contract like a rubber band. Instead, we could formulate the contour energy so that it is subjected to ballooning forces that make it expand. In this case, we can define an initial contour inside the object of interest and have it expand until it latches on to the boundary of the object.

### Medical Image Segmentation



27

### Interactive Image Segmentation



Magnetic Lasso Tool in Photoshop

28

Let us take a look at a couple of examples. On the left is a scan of a skull. Despite the complex boundary of the skull the contour (shown in red) is able to find it quite well. Contours have also been used to develop interactive image segmentation, such as the magnetic lasso tool in Photoshop. On the right a user is able to use this tool to find the boundary of the flower by roughly tracing it on the screen.

## Hough Transform

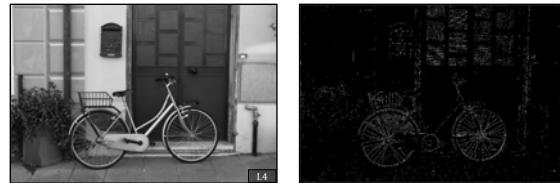
Shree K. Nayar

Columbia University

Topic: Boundary Detection, Module: Features  
First Principles of Computer Vision

29

## Difficulties for the Fitting Approach



- Extraneous Data: Which points to fit to?
- Incomplete Data: Only part of the model is visible.
- Noise

Solution: Hough Transform

30

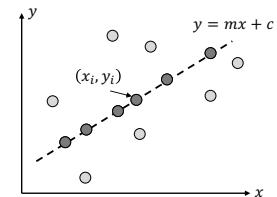
One of the key problems with boundary detection is knowing which edges in an image actually correspond to the boundary that we are looking for. If we knew that, then the problem would already be half-solved, and we can use one of the methods we have discussed above to find the boundary. Shown on the right is an image and its edge map. Imagine that we want to find the two wheels of the bicycle, which can be described as circles. Indeed, we can see the circles in the edge map, but we also see lots of other stuff. So, one challenge here is to be able to find the circles while being unaffected by all the other edges in the image. Another challenge is that of incomplete data. In the case of the bicycle, not all edges on the two wheels have been detected, creating some large gaps between the detected edges. In fact, the incomplete data problem could be even worse if the wheels are partially occluded by other objects in the scene. Finally, we have noise — some of the edges on the wheel may be detected away from the wheel, and there may be other edges close to the wheel that do not correspond to the wheel. The Hough transform, which was invented in the early 1960s, gives us a powerful way to deal with these three problems in one shot. It works very well for simple shapes that can be described using a small number of parameters.

Let us take a look at how the Hough transform works for the simplest of all shapes: the straight line. Shown here are a set of edges in an image and embedded in the set are points (dark dots) that lie on a straight line (dotted line). Our goal is to find the straight line from the complete set of points. The equation of the straight line is  $y = mx + c$ , where  $m$  is the slope and  $c$  is the y-intercept. If we consider one point on the straight line, for example  $(x_i, y_i)$ , we can write  $y_i = mx_i + c$ . Since  $x_i$  and  $y_i$  are known, we can rewrite this as a straight line equation in  $m$ - $c$  space, that is,  $c = -mx_i + y_i$ . This allows us to look at the problem in two spaces. One is the image space, which is the  $x$ - $y$  space, and the other is the parameter space, which is the  $m$ - $c$  space.

### Hough Transform: Line Detection

Given: Edge Points  $(x_i, y_i)$

Task: Detect line  
 $y = mx + c$



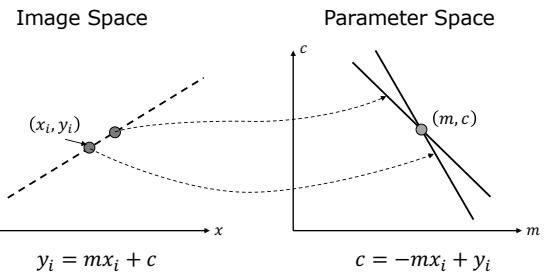
Consider point  $(x_i, y_i)$

$$y_i = mx_i + c \iff c = -mx_i + y_i$$

[Hough 1962] 31

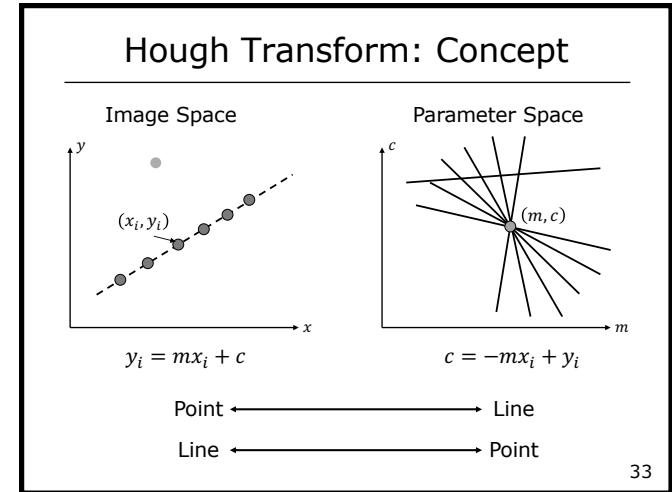
Now let us consider a line (shown as dotted) in image space and one point  $(x_i, y_i)$  that lies exactly on the line. We know this point corresponds to the straight line  $c = -mx_i + y_i$  in parameter space. This line in parameter space corresponds to all the lines that pass through the point  $(x_i, y_i)$  in image space. If we take another point in image space, it will give us another line in parameter space. Note that the two lines in parameter space intersect at a single point  $(m, c)$  which corresponds to the slope and intercept of the dotted line that passes through the two points in image space. In fact, if we take more points in image space that lie on the same straight line, we are going to get more lines in parameter space, but they are all going to intersect at the same point  $(m, c)$  since that is the only straight line that passes through all of them in image space.

### Hough Transform: Concept

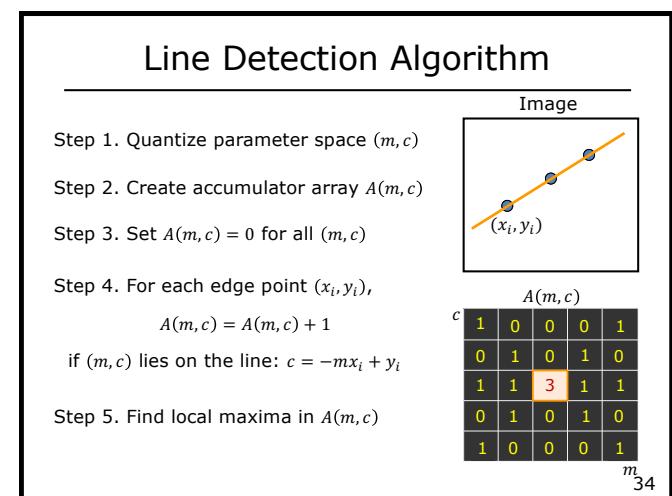


32

If we take a point that does not lie on the dotted line in image space, it too will create a line in parameter space. However, that line is not going to pass through  $(m, c)$  because the image point does not lie on the straight line with parameters  $(m, c)$ . This relationship between image space and parameter space is interesting. A point in image space maps to a line in parameter space, while a line in image space ends up as a point in parameter space. Based on this observation, we can create an algorithm to detect straight lines.

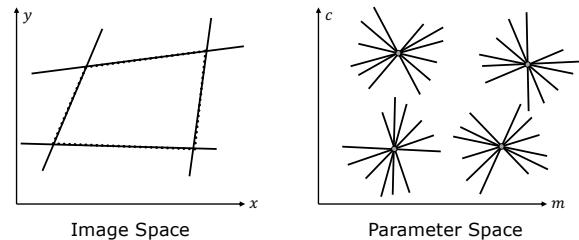


Detailed here is the Hough transform algorithm for line detection. Shown on the right is a very simple example image with just three points that lie on a line. First, we discretize our parameter space  $(m, c)$  using a resolution that suits our application. Next, we define a two-dimensional array of memory, called the accumulator array, to represent the discrete parameter space. The accumulator array is initialized with zeros. Now let us consider one of the edges in the image,  $(x_i, y_i)$ , and plug it into the line equation to get  $c = -mx_i + y_i$ , which we know is a line in the parameter space. Next, we increment every cell in the accumulator array that falls on this line. In effect, for each point in image space, we “vote” for points in parameter space. So, for our first image point, we get a line of 1’s in the accumulator array. We repeat this process of voting for the second image point. Where the two lines in parameter space intersect, we will get a 2 in the accumulator array, and all other cells in the array will be either a 0 or a 1. Finally, we take our third point, and vote along another line in the accumulator array. Note that the maximum value of 3 in the accumulator array corresponds to the  $m$  and  $c$  values of the line in image space we are looking for. If we have multiple lines in the image, we will get several local maxima in the accumulator array. The last step of line detection then is to simply find all the local maxima in the accumulator array.



In the example shown here, we have four line segments, which results in four intersections in parameter space. Note that the Hough transform allows us to find these four lines without giving care to which point in image space belongs to which line. We simply repeat our voting process for each point in image space and if a strong maximum emerges in parameter space, we know there are a significant number of edges in the image that lie on a line. This demonstrates the power of the Hough transform.

### Multiple Line Detection



35

When we use  $y = mx + c$  as the line equation, we know that  $m$  can range from negative infinity to positive infinity. This poses a practical problem; if we are interested in lines of all orientations, we would need a massive accumulator array. The solution is to use a different parametrization of the straight line, which we have used before. Here, the line parameters are  $\theta$  and  $\rho$ , where  $\theta$  must lie between 0 and  $2\pi$ , and  $\rho$  cannot be bigger than the size of the image itself, as it is the distance of the line from the origin of the image.

### Better Parameterization

Issue: Slope of the line  $-\infty \leq m \leq \infty$

- Large Accumulator
- More Memory and Computation

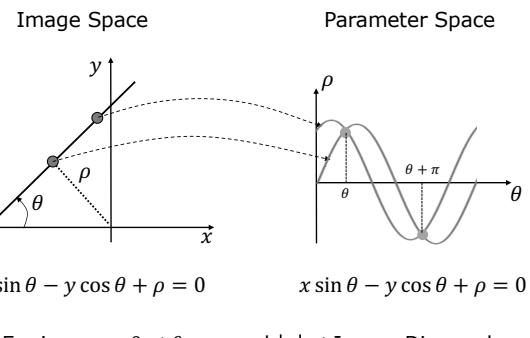
Solution: Use  $x \sin \theta - y \cos \theta + \rho = 0$

- Orientation  $\theta$  is finite:  $0 \leq \theta < \pi$
- Distance  $\rho$  is finite

36

Consider a single point in image space. With our new line parameterization, the point is going to map to a sinusoid in parameter space. Now, if we take another point in image space, it will map to another sinusoid in parameter space. The intersection of these sinusoids yields the parameters of the straight line on which the two image points lie. We see that we get two intersections here, where one of them corresponds to  $\theta$  and the other one corresponds to  $\theta + \pi$ , which is essentially the same straight line. So, in this case, the Hough transform uses an accumulator array with the parameters  $\rho$  and  $\theta$ , and voting is done along sinusoids.

### Better Parameterization



For images:  $0 \leq \theta < \pi$  and  $|\rho| \leq \text{Image Diagonal}$

37

Let us discuss some issues related to the Hough transform. The first is determining how big the cells of the accumulator array should be. Remember that if we make it a very low-resolution accumulator array (large cells), we could have a lot of votes falling within a cell, because there could be many lines with similar parameters that can pass through the same cell. Conversely, if the cells are too small, then, in the presence of noise, quantization, and other effects on the image side, we may not have any cell that gets a high enough number of votes to be declared a maximum. The second issue is related to finding the maxima after the voting process. After voting, we can expect the peaks in the array to be somewhat blurred due to image noise and the finite resolution of the array. To this end, we need to use some sort of a peak-finding algorithm, such as the non-maximal suppression method we used to find corners. Finally, since lines in the image are not expected to be perfectly straight, we are better off not voting for just a single cell at a time as we traverse a line in the accumulator array. Instead, it would make sense to vote for a small patch of cells, where the strength of the vote is maximum at the center and tapers off as we go to the edge of the patch.

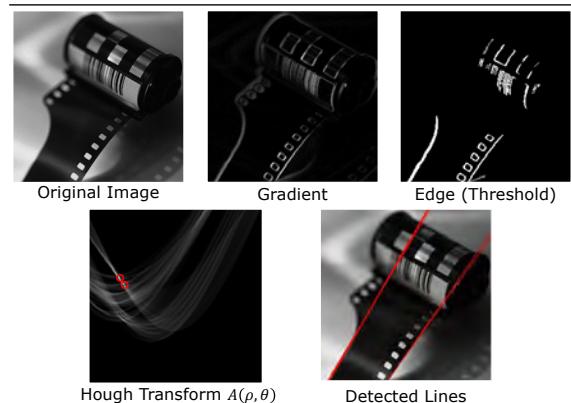
Let us take a look at how the Hough transform works on real images. Consider the original image on the upper left. To its right are the edge strength and a thresholded edge map. Here, we have used the  $\rho - \theta$  parametrization; note the sinusoidal streaks in the accumulator array. The array has two strong peaks, which correspond to the two strong lines (shown in red) overlaid on the original image in the bottom right.

## Hough Transform Mechanics

- How big should the accumulator cells be?
  - Too big, and different lines may be merged
  - Too small, and noise causes lines to be missed
- How many lines?
  - Count the peaks in the accumulator array
- Handling inaccurate edge locations:
  - Increment patch in accumulator rather than single point

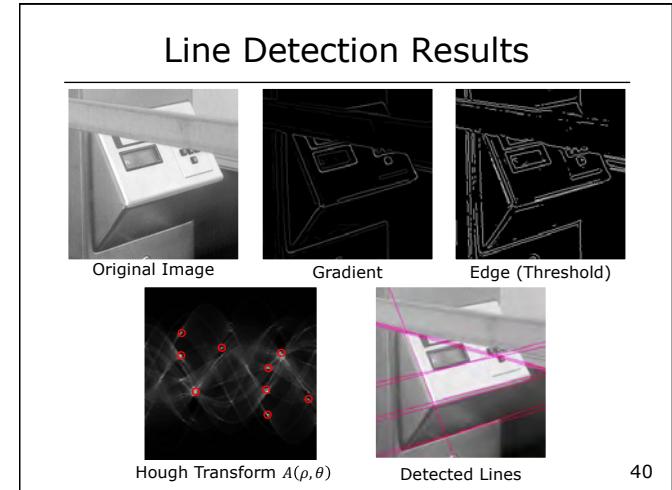
38

## Line Detection Results

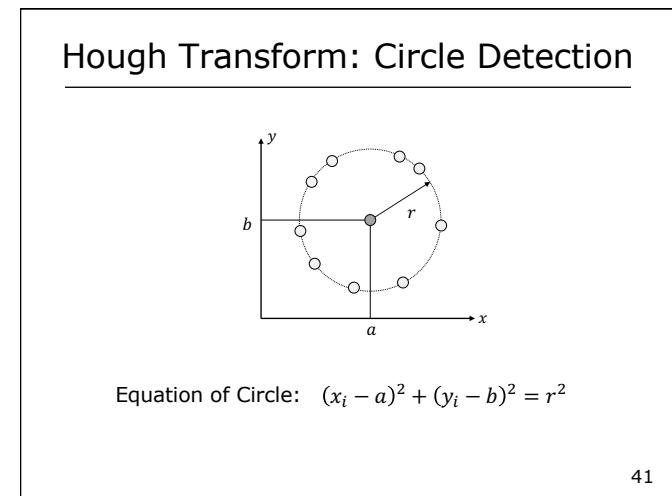


39

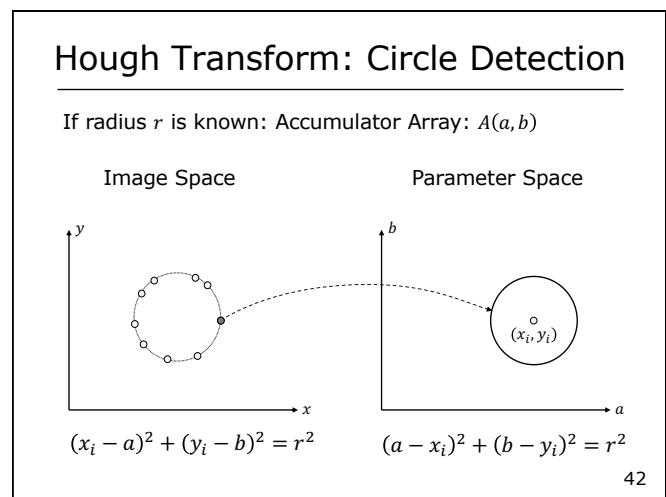
Here is a more complex image and its thresholded edge map. In this case, we get several strong peaks in the accumulator array. The detected lines are again overlaid on the original image in the bottom-right corner. We see that we detect some lines that are close to each other. This is because, in the presence of quantization and noise, we are going to end up detecting some peaks that are very close to each other.



We have seen how we can find lines using the Hough transform. Now let's take it up a notch and talk about how we can find circles. Here we have a set of points in the image. Below, is the general equation of a circle where the point  $(a, b)$  corresponds to the center of the circle and  $r$  is its radius. Given a set of edges, the three parameters we are interested in finding are  $a$ ,  $b$ , and  $r$ .



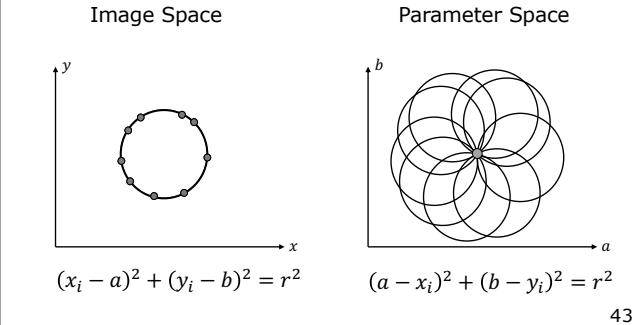
Let us make the problem easier by assuming that we know the radius  $r$  of the circle we are trying to find. In this case, we have a parameter space (accumulator array) with only two parameters,  $a$  and  $b$ . By rewriting the equation of the circle, we see that, given any image point  $(x_i, y_i)$ , we get the equation of a circle of radius  $r$  in the  $a$ - $b$  parameter space. What that means is that each point in the image will map to a circle in parameter space.



Consider this set of image points that lie on a circle of known radius. Each point will produce a circle in parameter space. All these circles will intersect at a single point which corresponds to the  $a$  and  $b$  values of the circle in the image that passes through all the image points. To find circles of a given radius therefore, we would vote along circles in the accumulator array. After the voting is done, peaks in the accumulator array would correspond to centers of detected circles of radius  $r$  in the image.

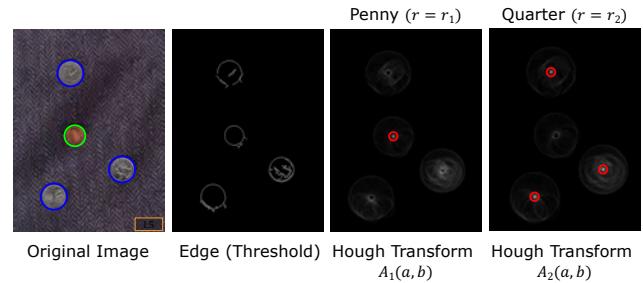
### Hough Transform: Circle Detection

If radius  $r$  is known: Accumulator Array:  $A(a, b)$



Now, let us take a look at a real image to which we will apply circle detection. The image includes three quarters and a penny. The penny, of course, has a different radius than the quarter. We apply edge detection to the image and the thresholded edges are seen in the second image. The last two images show the accumulator arrays after voting for circles with the radius of the penny and the quarter, respectively. Note that the strong peaks (in the red circles) in these two images do indeed correspond to the locations of the penny and the three quarters.

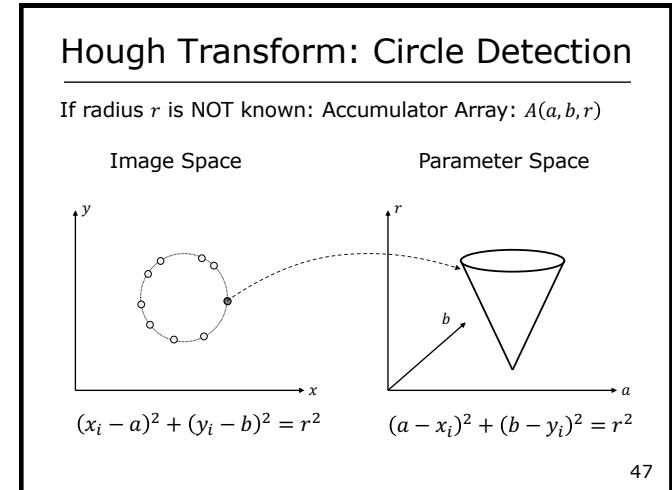
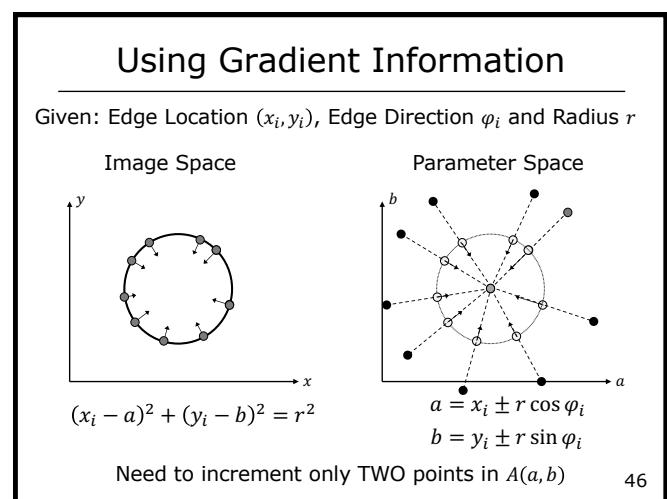
### Circle Detection Results



With the Hough transform, if we are willing to do more work in image space, we have less work to do in parameter space, and vice versa. To illustrate this, let's again consider the problem of finding circles of known radius,  $r$ . This time, though, we are given not only the location of each edge in the image, but also its direction,  $\varphi_i$ . Now, if we consider a single point in the image, we know that, if it lies on a circle of radius  $r$ , the center  $(a, b)$  of that circle must lie along the edge direction. We do not know on which side of the edge it lies, just that it lies along the edge direction at a distance  $r$  from the point itself. So, in parameter space, we no longer need to vote along an entire circle, we only have to vote at two points on either side of the edge. Thus, we only need to increment two points in the accumulator array, which is computationally a major saving. In other words, by doing the extra work of finding accurate edge directions in image space, we are able to significantly lower the work we need to do in parameter space.

Now let us take a look at a more interesting case, where the radius of the circle is not known. We are also going to assume that we do not know the direction of each edge. Now, we have three unknowns in our equation for the circle —  $a$ ,  $b$ , and  $r$  — and so our parameter space, and hence the accumulator array, will have to be three-dimensional. For any given edge location  $(x_i, y_i)$  we will need to vote over an entire surface in the three-dimensional accumulator array. If we take a close look at the equation of the surface shown on the right, we see that it is the equation of a cone.

So, for each point we need to vote along a cone, and when we are done voting for all image edges, we look for the maxima in the array. This simple example illustrates that the work that needs to be done in parameter space increases exponentially with the number of unknown parameters. In short, as the parametric shape we are looking for increases in complexity, the Hough transform becomes less and less practical.



## Generalized Hough Transform

Shree K. Nayar

Columbia University

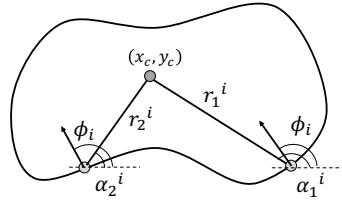
Topic: Boundary Detection, Module: Features and Boundaries

First Principles of Computer Vision

48

## Generalized Hough Transform

Find shapes that cannot be described by Equations



Reference point:  $(x_c, y_c)$

Edge direction:  $\phi_i \quad 0 \leq \phi_i < 2\pi$

Edge location:  $\vec{r}_k^i = (r_k^i, \alpha_k^i)$

[Ballard 1981]

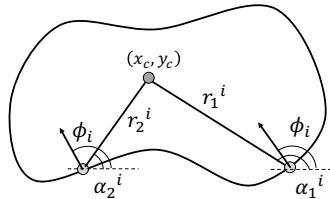
49

We have seen how we can use the Hough transform to find simple shapes that can be described with a small number of parameters. Now let us take a look at how we can generalize the Hough transform to find more complex shapes that cannot be described using an equation. This is called the Generalized Hough Transform (GHT). Given a shape such as the one shown above, the first thing we are going to do is define a reference point  $(x_c, y_c)$  for the shape. It turns out that when we apply the GHT to find an object, we are going to be voting for the location of its reference point. In other words, our accumulator array is two-dimensional and has the parameters  $x_c$  and  $y_c$ . If we get a strong peak in the array, then we have found the object and its location in the image is determined by the location of its reference point.

The first step is to create a model of the object that can be used by the Hough transform. This step is done off-line. For each point on the object's boundary, we are going to assume that we have both the edge location and the edge direction,  $\phi$ . Then, for each point boundary point, we represent it using the vector  $\vec{r}$ , which includes the distance  $r$  of the point from the reference point and the angle  $\alpha$  the edge makes with respect to the horizontal axis.

We use the above approach to create a model of the object in the form of a table, called the  $\phi$ -table. The index to the table is the edge direction  $\phi_i$  and the entry is a list of the vectors  $\vec{r}_k^i$  corresponding to all the points on the object's boundary that have that edge direction  $\phi_i$ . This table is the model that we will use to perform Hough transform and find the object in an image. Note that the assumption here is that the object should appear in the image with the same orientation and scale. It can, however, appear

## Hough Model



$\phi$ -Table:	Edge Direction	$\vec{r} = (r, \alpha)$
	$\phi_1$	$\vec{r}_1^1, \vec{r}_2^1, \vec{r}_3^1$
	$\phi_2$	$\vec{r}_1^2, \vec{r}_2^2$
	$\vdots$	$\vdots$
	$\phi_n$	$\vec{r}_1^n, \vec{r}_2^n, \vec{r}_3^n, \vec{r}_4^n$

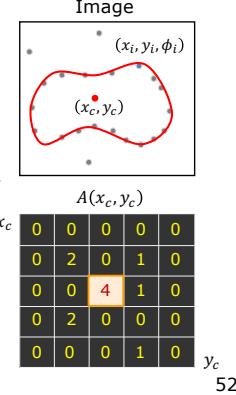
50

anywhere in the image, have missing boundary data, or even be partially occluded.

Here is an outline of how the generalized Hough transform uses the above object model to find it in an image. In the image shown here, we can see that points belonging to the above object are present, along with other points. Our goal is to find the location of the reference point  $(x_c, y_c)$  if, indeed, the object lies in the image. We begin by creating an accumulator array,  $A(x_c, y_c)$ , and initialize it to zero. Remember that for each point, we have both the location and the direction of the edge at that point. So, we use the edge direction  $\phi_i$  of the point as an index into our  $\phi$ -table to find all the vectors  $\vec{r}_k^i$  associated with it. We use the vectors to vote for the reference point in the accumulator array. Once we have voted using all the edges in the image, if a strong peak emerges in accumulator array, we have found an instance of the object such that the location of the reference point corresponds to the peak.

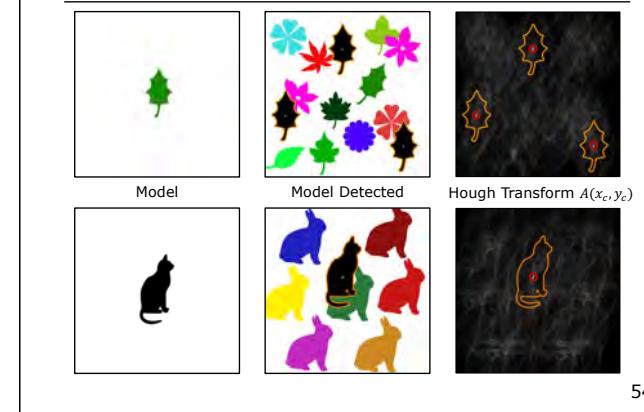
### Generalized Hough Algorithm

- Create accumulator array  $A(x_c, y_c)$
  - Set  $A(x_c, y_c) = 0$  for all  $(x_c, y_c)$
  - For each edge point  $(x_i, y_i, \phi_i)$ ,
- For each entry  $\phi_i \rightarrow \vec{r}_k^i$  in  $\phi$  - table,
- $$x_c = x_i \pm r_k^i \cos(\alpha_k^i)$$
- $$y_c = y_i \pm r_k^i \sin(\alpha_k^i)$$
- $$A(x_c, y_c) = A(x_c, y_c) + 1$$
- Find local maxima in  $A(x_c, y_c)$



Let us see the result of applying the GHT to some simple images. Shown on the left are images of the objects — a leaf and a cat — we are interested in detecting. We use these images to create  $\phi$ -tables for each of the two objects. Then, we apply the GHT to the cluttered images shown in the middle column. On the right we see the strong peaks (highlighted by the red circles) in the accumulator array. For each strong peak, the outline of the object is overlaid to show that the objects and their locations are correctly detected. Note that in the case of the leaf, GHT is able to find it even when it is partially occluded.

### Results



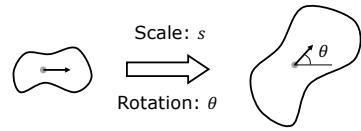
Could we modify the above algorithm so that it can detect the objects in arbitrary scale and rotation? In principle, this is possible. As shown here, we can simply incorporate the unknown scale and rotation by adding them as parameters to our accumulator array. In doing so, we go from a two-dimensional accumulator array to a four-dimensional array. Given an edge in the image we can use the modified equations here to vote in the accumulator array. Each strong peak in the array corresponds not only to the existence of the object in the image, but also reveals its scale and rotation.

However, the memory required for the four-dimensional array and the computational cost of voting within it makes such an approach impractical in most real-world applications.

We will conclude with a few remarks related to the Hough transform. First, the Hough transform is very attractive as it works on disconnected edges; as long as we have enough edges that actually lie on the boundary of the object, we are not going to be affected by noise in the image, extraneous edges, or even partial occlusion of the object. It works extremely well for simple shapes with a small number of parameters, such as lines and circles. It can also work on complex shapes using the generalized Hough transform, but it requires the edges of the object to be detected with high accuracy in terms of both location and direction. Finally, there is a trade-off between the work that we are doing in our image space and the work that we need to do in our parameter space. As we showed earlier, if edge orientation can be computed with high accuracy, we can significantly reduce the number of votes needed in parameter space.

In summary, the Hough transform is an elegant and useful technique for detection problems that involve shapes that can be well-described with a small number of parameters.

### Handling Scale And Rotation



Use Accumulation Array:  $A(x_c, y_c, s, \theta)$

$$x_c = x_i \pm r_k^i \cdot s \cos(\alpha_k^i + \theta)$$

$$y_c = y_i \pm r_k^i \cdot s \sin(\alpha_k^i + \theta)$$

$$A(x_c, y_c, s, \theta) = A(x_c, y_c, s, \theta) + 1$$

Huge Memory and Computationally Expensive!

55

### Hough Transform: Comments

- Works on disconnected edges
- Relatively insensitive to occlusion and noise
- Effective for simple shapes (lines, circles, etc.)
- Complex Shapes: Generalized Hough Transform
- Trade-off between work in image space and parameter space

56

## References and Credits

Shree K. Nayar

Columbia University

Topic: Boundary Detection, Module: Features  
First Principles of Computer Vision

57

## References: Papers

[Ballard 1981] D. H. Ballard. "Generalizing the Hough Transform to Detect Arbitrary Shapes". *Pattern Recognition*, vol. 13, no.2, 1981.

[Duda and Hart 1975] R. O. Duda and P. E. Hart. "Use of the Hough Transform to Detect Lines and Curves in Pictures". *Comm. ACM*, vol.15, 1975.

[Hough 1962] P. V. C. Hough. *Method and Means for Recognizing Complex Patterns*. U.S. Patent 3069654, 1962.

[Kass 1987] M. Kass, A. Witkin and D. Terzopoulos. "Snakes: Active Contour Models", *IJCV*, 1987.

[Xu 1997] C. Xu and J. Prince. "Gradient Vector Flow: A New external force for Snakes", *CVPR*, 1997.

58

## Image Credits

I.1 Tolga Birdal. Used with permission.

I.4 Purchased from iStock by Getty Images.

I.5 Vivek Kwatra. Used with permission.

59

---

Acknowledgements: Thanks to Nisha Aggarwal and Jenna Everard for their help with transcription, editing and proofreading.

## References

[Nalwa 1994] A Guided Tour of Computer Vision, Nalwa, V., Addison-Wesley, 1993.

[Ballard 1981] D. H. Ballard. "Generalizing the Hough Transform to Detect Arbitrary Shapes". *Pattern Recognition*, vol. 13, no.2, 1981.

[Duda and Hart 1975] R. O. Duda and P. E. Hart. "Use of the Hough Transform to Detect Lines and Curves in Pictures". *Comm. ACM*, vol.15, 1975.

[Hough 1962] P. V. C. Hough. *Method and Means for Recognizing Complex Patterns*. U.S. Patent 3069654, 1962.

[Kass 1987] M. Kass, A. Witkin and D. Terzopoulos. "Snakes: Active Contour Models", *IJCV*, 1987.

[Xu 1997] C. Xu and J. Prince. "Gradient Vector Flow: A New external force for Snakes", CVPR, 1997.

[Nayar 2022E] [Image Processing I](#), Nayar, S. K., Monograph FPCV-1-4, First Principles of Computer Vision, Columbia University, New York, March 2022.

[Nayar 2022F] [Image Processing II](#), Nayar, S. K., Monograph FPCV-1-5, First Principles of Computer Vision, Columbia University, New York, March 2022.

[Nayar 2022G] [Edge Detection](#), Nayar, S. K., Monograph FPCV-2-1, First Principles of Computer Vision, Columbia University, New York, May 2022.

[Nayar 2022H] [Boundary Detection](#), Nayar, S. K., Monograph FPCV-2-2, First Principles of Computer Vision, Columbia University, New York, June 2022.

[Nayar 2025M] [Image Segmentation](#), Nayar, S. K., Monograph FPCV-5-2, First Principles of Computer Vision, Columbia University, New York, May 2025.