EE 113D: Digital Signal Processing Design

Mini-Project #1
Object Detection

Names: Erik Hodges & Ivan Manan

Due Date: November 14, 2018

Objective

The purpose of this project was to identify images by their shapes and orientation via the Hough Transform.

Theory

The Hough transform maps every black pixel in an image to a set of lines that pass through that point. Consequently, points that lie on a common line will all map to that line, creating a peak in the Hough space corresponding to that line. Each line is characterized by its minimum distance from the origin, which we call *r*, and the angle that an orthogonal line that passes through the origin makes with the x-axis, which we call theta. By looking at the peaks of the Hough transform, we can reconstruct simple shapes by drawing the lines with the associated *r* and theta values.

We map a certain point in the image to a set of lines by predefining a set of theta values and calculating the associated r values for each angle via the following equation:

$$r = y \sin\theta + x \cos\theta$$

Each pair of r and theta values is cast into the Hough space as one vote for the given line.

Design Rationale

For our design, we chose to increment r at intervals of one pixel and increment theta at intervals of 15 degrees. Theta ranges from 0 to 180 degrees. We also had to calculate the full range of *r* values. The range of positive r values is given by the pythagorean sum of the width and height of the image:

$$Rmax = \sqrt{width^2 + height^2}.$$

Looking at the equation for r:

$$r = y \sin\theta + x \cos\theta.$$

We note that theta is contained on the interval [0, 180] and

$$\sin\theta > 0 \ \forall \ \theta \in [0,\ 180].$$

Therefore, since the minimum value of $\cos\theta$ is -1 on this interval,

$$Rmin = -width.$$

We calculate the number of bins for r values in the Hough transform array by:

$$RBINS = -Rmin + Rmax.$$

The number of bins for theta is

$$TBINS = \frac{180}{15} + 1 = 13.$$

Because RBINS is dependent on the size of the image and C requires all array sizes to be declared at compile time, we used pointers to create a dynamic array that contains the bins of the hough transform. One challenge was properly indexing the array since it is not feasible to make a 2D dynamic array in C. The format of the array is shown below

		Theta (deg)						
		0	15	30	...	150	165	180
r	Rmin	0	1	2	...	10	11	12
	Rmin+1	1*13 + 0	1*13 + 1	1*13 + 2	...	1*13 + 10	1*13 + 11	1*13 + 12
	Rmin+2	2*13 + 0	2*13 + 1	2*13 + 2	...	2*13 + 10	2*13 + 11	2*13 + 12
	...	...	...	...	...	...	...	...
	Rmax-1	(RBINS-2) *13 + 0	(RBINS-2) *13 + 1	(RBINS-2) *13 + 2	...	(RBINS-2) *13 + 10	(RBINS-2) *13 + 11	(RBINS-2) *13 + 12
	Rmax	(RBINS-1) *13 + 0	(RBINS-1) *13 + 1	(RBINS-1) *13 + 2	...	(RBINS-1) *13 + 10	(RBINS-1) *13 + 11	(RBINS-1) *13 + 12

Table 1: Dynamic Array Indexing. The values in the grid are the array indices for the associated r-theta pairs. Each consecutive set of 13 entries corresponds to the 13 theta values that are paired with a single r value.

To optimize our code, we created lookup tables for the sine and cosine functions so that we only have to call the sin(x) and cos(x) functions (which are slow) once. This was done in the fill_table() function.

The hough_transform function takes in the x and y coordinates of each black pixel in the image, computes the 13 r-theta pairs associated with that point, and adds votes for the pairs to the hough dynamic array.

Data

The Hough transformed mapped the bitmap data into theta θ and distance r values. The theta values were fixed by splitting 180 degrees into 13 values with 15 degree increments. The purpose of fixing the theta θ values was for simplicity sake and that we were not given complicated polygons; therefore, we did not need accurate precision with the angles. The distance r values was calculated from the equation as aforementioned in the Theory section by using fixed theta values and every (x, y) pair values.

Consider Figure 1. The image of a rectangle was given as input and processed via its bitmap. The Hough transform in Figure 1 depicts a pair of peaks at 90 degrees which correspond to the horizontal sides of the rectangle. Moreover, there are two pairs of peaks at 0 and 180 degrees which correspond to the vertical sides of the rectangle. Any peaks at 0 degrees have an associated adjacent peak at 180 degrees since lines from the origin angled at 0 and 180 are collinear. The pairs of peaks at 0 and 180 degrees have r values with equal magnitudes but opposite signs. Also note that, since the vertical sides of the rectangle are longer, the peaks at 0 and 180 degrees are higher than the peaks at 90 degrees. Since the peaks correspond to two horizontal lines and two vertical lines, it can be deduced that the original image is a rectangle with sides parallel to the horizontal and vertical axes.

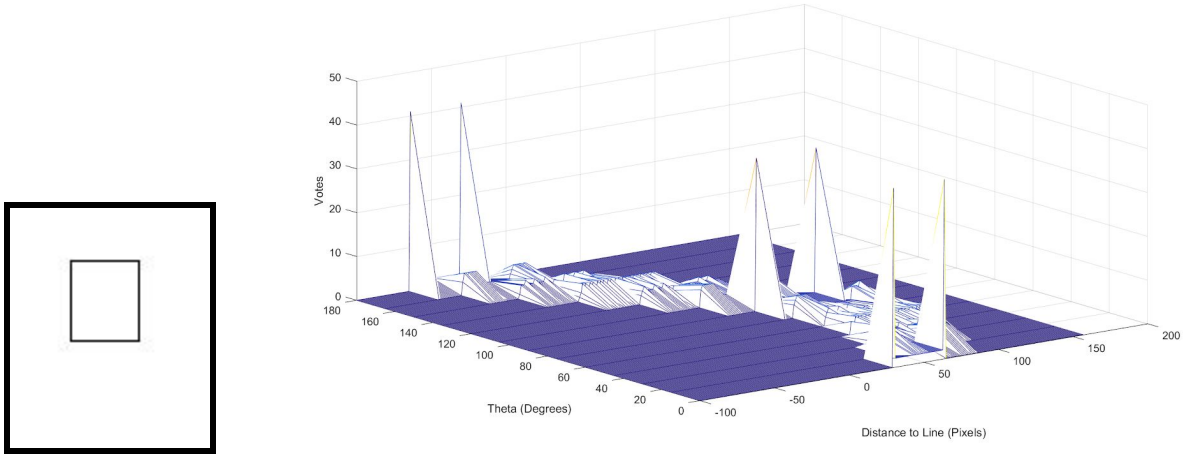

Figure 1. A Square Image (Left) with its Hough Transform (Right)

Consider Figure 2. The input is a rectangle similar to the input in Figure 1, but with a different orientation. There are four peaks, one pair of peaks at 45 degrees and another pair of peaks at 135 degrees. The peaks at 135 degrees are higher because they correspond to the longer line segments that have an angle of 45 degrees in the original image. Since the peaks correspond to two 45 degree lines and two 135 degree lines, and the angle between these lines is 90 degrees, it can be deduced from the Hough transform that the original image is a rectangle that is rotated by 45 degrees.

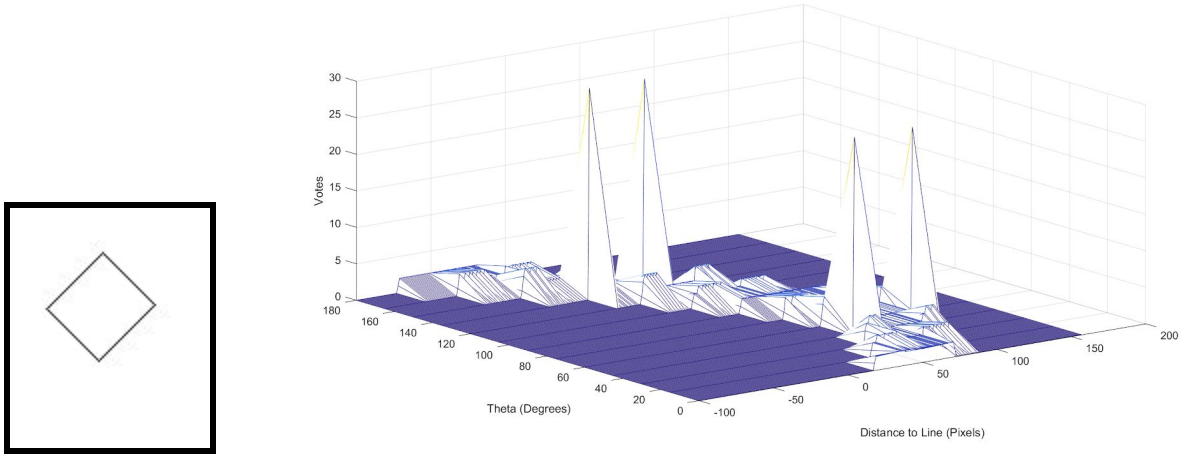

Figure 2. A Diamond Image (Left) with its Hough Transformation (Right)

Finally, consider the hexagonal image and its respective Hough transform in Figure 3. Given that the lines in the image have angles of 30 degrees, 150 degrees and vertical, we would expect to see peaks in the Hough space at 0, 60, 120, and 180 degrees. The Hough transform below supports this assumption. However, since the hexagon has equal side lengths, we should also expect expect each peak to be roughly the same height, but the peaks at 0 and 180 degrees are much higher than the others. A possible explanation of this is that, since the image is composed of pixels, the diagonal sides of the hexagon are actually composed of a number of short horizontal lines. Because of this, various pixels on the diagonal lines had r values that differed by one or two pixels from the true r value, resulting in shorter peaks. Since the peaks in the hough space correspond to 6 lines, two of which are vertical, we can deduce from the hough transform that the original image is a hexagon with two vertical sides.

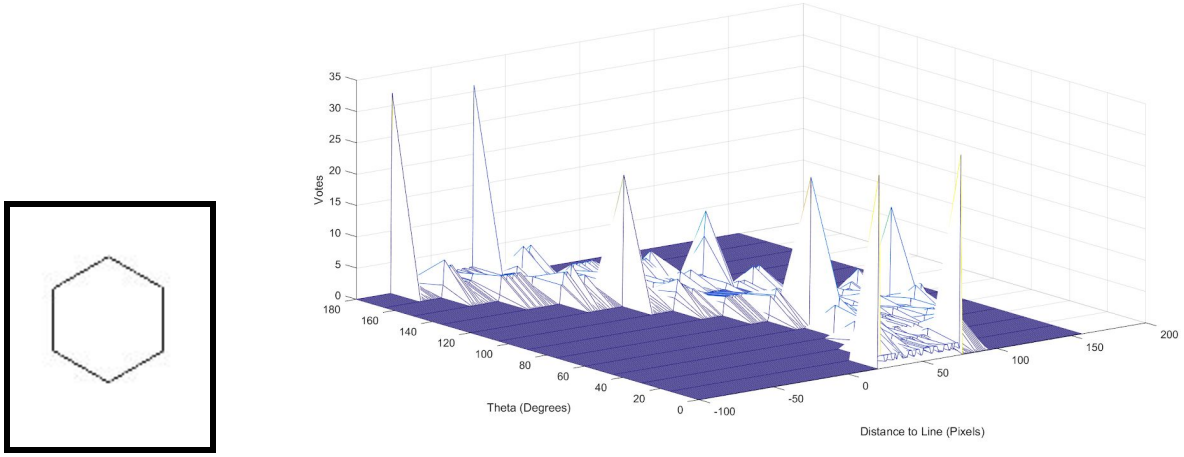

Figure 3. A Hexagonal Image (Left) with its Hough Transformation (Right)

Figures 1, 2, and 3 had the Hough transform done on known inputs. Consider Figure 4, which displays only the Hough transformation. The input image is unknown. However, by analyzing the Hough transformation, we are able to attain the original input. In Figure 4, the peaks are located at 0, 45, and 90 degrees. Taking the angles and specific distance values, we are able to sketch a right triangle in quadrant 1 of a Cartesian coordinate plane, with the legs of the right triangle parallel to the x and y axes. The hypotenuse has a negative slope.

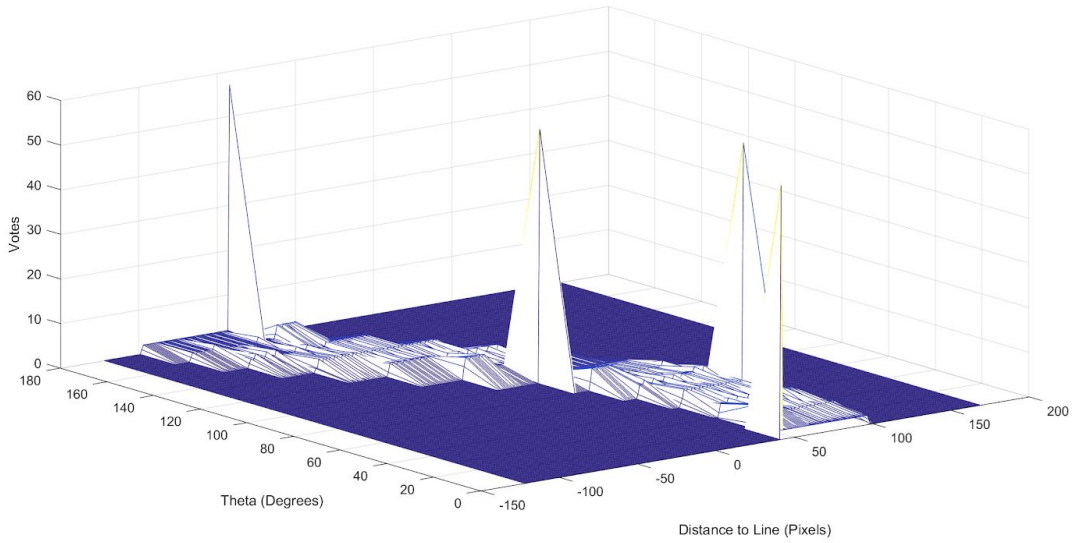

Figure 4. The Hough Transformation of an Unknown Image

Challenges

The main challenges we had was solving the indexing errors with Matlab and constructing pointers to emulate dynamically allocated arrays. The issue with MATLAB when calling the `reshape` function was that the indices of the array started off at 1 instead of 0. This was problematic because the C code starts the indices of arrays at 0. As a result, an off-by-one error persisted in Matlab. The way to overcome this error was to declare new x and y values that properly maps the theta and distance values to the correct coordinates of the number of votes.

The last challenge was resolving errors with pointers in C. Pointers were used in order to create dynamically allocated arrays because the size of the input image would always vary. Therefore, hard-coding and manually calculating the size of the input image and its respective bitmap is tedious. The main challenges that persisted with pointers was that it was difficult to deduce whether correct values were properly inputted into the memory address. As a result, precise mathematical measurements were done to make sure all the indices and distance values were accurate. Moreover, we had to account for the memory address of certain variables, such as the memory address of the Hough transform array. We had to go into the memory browser to look

up the memory address of the Hough transform array in order to export its values to Matlab using the save memory function.

Name Erik Hodges

Name Ivan Manan

Successfully completed Mini-Project 1. ✓

Successfully completed Extra Credit (1/3 slower playback)

Comments

Signed Dill Briggs

Code for Hough Transformation

```
// Infoheader.width --> bitmap
// Infoheader.height --> bitmap

// Declare global variables here

#define PI 3.14159265
int* hough;

#define TBINS 13 //13 angles at 15 deg increments
float sine_table[TBINS];
float cos_table[TBINS];



void fill_table() {

	//create lookup table for sine and cosine
	int h;
	float val = 15 * PI / 180;
	for (h = 0; h < TBINS; h++) {
		sine_table[h] = (float)(sin(h * val));
		cos_table[h] = (float)(cos(h * val));
	}
}

void hough_transform(int x, int y, int* hough_out, int x_max) {

	int i;
	float sin_theta, cos_theta;
	float r;
	int radius;
	for (i = 0; i < TBINS; i++) {
		sin_theta = sine_table[i];
		cos_theta = cos_table[i];

		// Round r to nearest integer
		r = x * cos_theta + y * sin_theta;

		radius = (int)round(r);

		//update hough transform array
		int h = hough[TBINS * (radius + x_max) + i];
		++h;
		hough[TBINS * (radius + x_max) + i] = h;

	}
}



int
main(void)
{
	fill_table();

	msc_inti();

	//
	//	Read a BMP file from the mass storage device
	//
	int i,j;
	mem_init();
```

```
	// Read Image from USB as a 2-d matrix and store it in "image"
	bitmap = usb_imread("Im01.bmp");
	image = m_malloc(InfoHeader.Height*InfoHeader.Width*sizeof(unsigned char));
	for(i = 0; i< InfoHeader.Height; i++)
		for(j = 0;j < InfoHeader.Width; j++)
			if(bitmap[(i*InfoHeader.Width+j)*3] < 10)
				image[i*InfoHeader.Width+j] = 1;
			else
				image[i*InfoHeader.Width+j] = 0;

	//calculate range of r values based on image dimensions
	const int RBINS = (InfoHeader.Width + (int)(sqrt(InfoHeader.Width * InfoHeader.Width
+ InfoHeader.Height * InfoHeader.Height) + 1));
	hough = m_malloc(RBINS*TBINS*sizeof(int)); //theta increment = 15 deg, r increment =
1 pixel

//allocate memory for the hough transform array

	//print out dimensions
	const int xdim = InfoHeader.Width;
	const int ydim = InfoHeader.Height;
	printf("%s", "Image Dimensions: ");
	printf("%d", xdim);
	printf("%s", "x");
	printf("%d\n", ydim);
	printf("%s", "RBINS = ");
	printf("%d\n", RBINS);

	// image prints 1's and 0's that displays the segment
	// bitmap is the actual array we will be using


////////////////////////////////////////////////////////////////////////////////////////
///
	// My Code

	// Initialize hough array as all zeros
	int r;
	for (r = 0; r < RBINS*TBINS; ++r) {
		hough[r] = 0;
	}


	// Do Hough transform
	for(i = 0; i< InfoHeader.Height; i++){
		for(j = 0;j < InfoHeader.Width; j++) {
			if(image[i*InfoHeader.Width+j] == 1) {
				// Transform x and y values
				hough_transform(j, i, hough, (int)InfoHeader.Width);
			}
		}
	}

	//Free memory
	m_free(bitmap);
	m_free(image);
	free(hough);

}
```