Background Information

In this lab exercise, you will be optimizing a kernel for an FPGA to calculate the Hough Transform of the pixels within a picture. You will follow the flow presented in class to achieve this optimization – you will verify functionality using emulation, and you will use an HTML optimization report to decipher which optimizations might be beneficial.

The Hough transform is used in computer vision applications. After an image has been processed with an edge-detection algorithm such as a Sobel filter, you are left with a monochrome (black/white) image. It is useful for many further detection algorithms to consider the image as a set of lines. However, an image of black and white pixels is not a convenient or useful representation of these lines to algorithms such as object detection. The Hough transform is a transform from pixels to a set of "line votes."

Before getting to the code, here is the theory behind the Hough Transform.

It is commonly known that a line can be represented in a slope-intercept form:

$$y = mx + b$$

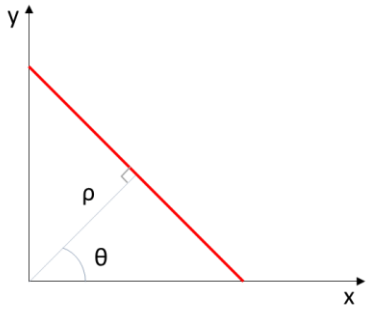
In this form, each line can be represented by two unique constants, the slope (m) and the y-intercept (b). So, every (m,b) pair represents a unique line. However, this form presents a couple of problems. First, since vertical lines have an undefined slope, it cannot represent vertical lines. Secondly, it is difficult to apply thresholding techniques to.

Therefore, for computational reasons in many detection algorithms, the Hesse normal form is used. This form has the equation below.

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

In this form, each unique line is represented by a pair (ρ,θ) (pronounced "rho" and "theta"). This form has no problem representing vertical lines, and you will learn how thresholding can easily be applied after the Hough Transform is applied.

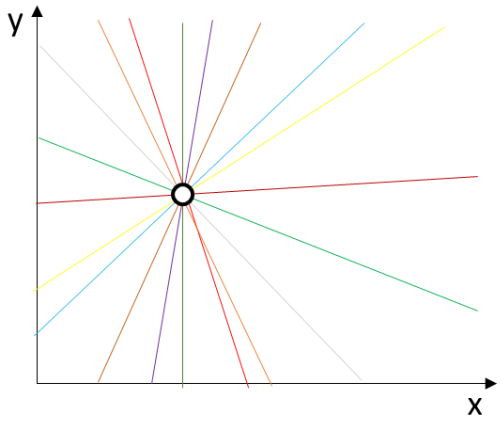
The following picture depicts what the ρ and θ values represent in the equation. For every line you want to represent (see the red line in the picture), there will be a unique line you can draw from the origin to it with the shortest distance (see the gray line in the picture). Another way of looking at this, is perpendicular line from the red line crossing the origin. ρ is the distance of the shortest line that can be drawn from the origin to the line you are wanting to represent. θ is the angle from the x-axis to that line.



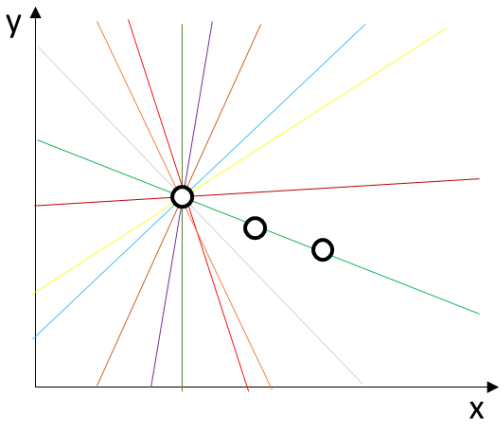
When working with an image, a corner of the image is traditionally considered to be the origin (the origin is not at the center), so the largest value ρ can be is the measure of the diagonal of the image. You can choose for the ρ values to all be positive, or to be allowed to be positive and negative. If you choose for all ρ values to be positive, then the range of θ is from 0 to 360 degrees. If you choose for ρ to be allowed to be positive or negative, the range of θ is from 0 to 180 degrees. These ranges are quantized in order to define a finite solution space.

Remember that the picture before being input into the Hough Transform has already gone through an edge detection algorithm and is therefore monochrome (each pixel is either black or white). The edges that have been detected are represented by the white pixels. The Hough Transform will transform the white pixels into an array of votes for lines.

Each white pixel in the image is potentially a point on a set of lines. The picture below represents the lines one pixel can potentially be a part of. (Note: not all potential lines are drawn, it is meant only for illustrative purposes.) A line with every potential slope passing through that pixel is potentially a line that appears in the image. So, 1 vote will be accumulated for each of these lines.



The code will loop through each pixel in the image, accumulating votes for lines as it goes. As a line accumulates more votes, the likelihood of that being a correct representation for a line in the picture goes up. So, as visualized below, the green line is going to accumulate 3 votes, which will make it a more likely candidate than the other lines. A threshold can easily be applied, therefore, by simply defining the amount of votes which is “enough” to define whether a line is present or not.



Now, let's take a look at the code to implement this transform. First, the complete algorithm will be shown, and then it will be explained piece by piece.

```
//A lookup table of sin and cos values at whole integer degree values
#include "sin_cos_values.h"

char pixel_array[IMAGE_HEIGHT*IMAGE_WIDTH];
short accumulators[THETAS*RHOS*2];

for (uint y=0; y<IMAGE_HEIGHT; y++) {

      for (uint x=0; x<IMAGE_WIDTH; x++){

            unsigned short int increment = 0;

            if (pixel_array[(WIDTH*y)+x] != 0) {
                  increment = 1;
            } else {
                  increment = 0;
            }

            for (int theta=0; theta<THETAS; theta++) {
                  int rho = x*cos_table[theta] + y*sin_table[theta];
                  accumulators[(THETAS*(rho+RHOS))+theta] += increment;
            }
      }
}
```

Let's first take a look at the array declarations at the top of the code.

```
char pixel_array[IMAGE_HEIGHT*IMAGE_WIDTH];
short accumulators[THETAS*RHOS*2];
```

The pixel array is the image itself, and each pixel occupies a place in the array.

The accumulators array will keep track of our line votes. Each place in the array represents a potential line in the image. Recall that a unique line is represented by a pair (ρ,θ). Therefore, the number of all of the potential lines in our image is equal to all potential values of ρ times all potential values of θ. ρ is the distance from the origin, which is defined as a corner of the picture. The greatest value of ρ is the measure of the diagonal of the picture. We will also let ρ be either positive or negative so that θ is bounded between 0 and 180 degrees. We will quantize at integer values for ρ, and integer degrees for θ. Our number of accumulators, therefore, is the measure of the diagonal of the picture (RHOS in the code) times 2 times 180 degrees (THETAS in the code).

Now, let's examine at the code to implement the algorithm.

```
for (uint y=0; y<IMAGE_HEIGHT; y++) {
      for (uint x=0; x<IMAGE_WIDTH; x++){
            …
      }
}
```

The outer loop will loop through each pixel in the image, accumulating votes for all the potential lines a pixel could be a part of as it goes.

```
unsigned short int increment = 0;

if (pixel_array[(WIDTH*y)+x] != 0) {
      increment = 1;
} else {
      increment = 0;
}
```

If the pixel is white (!=0), then we will add a 1 to all of the accumulators for potential lines defined by that pixel. Otherwise, we will not add anything to the accumulator. We do it in this manner so that the control logic inside the FPGA and the computation logic that we will duplicate later with pragmas is simpler and consumes less logic resources..

```
for (int theta=0; theta<THETAS; theta++) {
      int rho = x*cos_table[theta] + y*sin_table[theta];
      accumulators[(THETAS*(rho+RHOS))+theta] += increment;
}
```

For each pixel location, all of the lines that can have that pixel location as part of their values needs to receive a vote. Recall the formula we are using to represent a line:

$$\rho = x\cos\theta + y\cos\theta$$

The x and y values are constants for the duration of this inner loop. We will plug in every possible value of θ, along with the x and y, and solve for ρ given that θ. We will then cast a vote (add 1 to the accumulator location) for that (ρ,θ) pair. In this manner, we cast a vote for every line that is a possibility, traversing the possibilities in an arc from 0 degrees to 180 degrees.

Now, let's begin the lab and see what optimizations we can do to improve the total execution time in the FPGA. Don't worry if you don't completely understand the algorithm, it is sufficient to think of it as a convenient piece of code to perform optimizations on. However, if you would like to learn more, the Wikipedia* entry for the Hough Transform is a great place to start:
https://en.wikipedia.org/wiki/Hough_transform