

FEATURE DETECTION IN AN INDOOR ENVIRONMENT USING HARDWARE ACCELERATORS FOR  
TIME-EFFICIENT MONOCULAR SLAM

By  
Shivang Vyas

A Thesis  
Submitted to the Faculty  
Of the  
WORCESTER POLYTECHNIC INSTITUTE  
In partial fulfillment of the requirements for the  
Degree of Master of Science  
In  
Electrical and Computer Engineering  
By

---

OCTOBER 2015

APPROVED: 

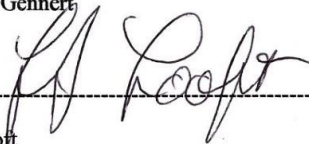
---

Professor William Michalson



---

Professor Michael Gennert



---

Professor Fred Looft

FEATURE DETECTION IN AN INDOOR ENVIRONMENT USING HARDWARE ACCELERATORS FOR  
TIME-EFFICIENT MONOCULAR SLAM

By  
Shivang Vyas

A Thesis  
Submitted to the Faculty  
Of the  
WORCESTER POLYTECHNIC INSTITUTE  
In partial fulfillment of the requirements for the  
Degree of Master of Science  
In  
Electrical and Computer Engineering  
By

-----  
OCTOBER 2015

APPROVED:

-----  
Professor William Michalson

-----  
Professor Michael Gennert

-----  
Professor Fred Looft

## Abstract

In the field of Robotics, Monocular Simultaneous Localization and Mapping (Monocular SLAM) has gained immense popularity, as it replaces large and costly sensors such as laser range finders with a single cheap camera. Additionally, the well-developed area of Computer Vision provides robust image processing algorithms which aid in developing feature detection technique for the implementation of Monocular SLAM. Similarly, in the field of digital electronics and embedded systems, hardware acceleration using FPGAs, has become quite popular. Hardware acceleration is based upon the idea of offloading certain iterative algorithms from the processor and implementing them on a dedicated piece of hardware such as an ASIC or FPGA, to speed up performance in terms of timing and to possibly reduce the net power consumption of the system. Good strides have been taken in developing massively pipelined and resource efficient hardware implementations of several image processing algorithms on FPGAs, which achieve fairly decent speed-up of the processing time. In this thesis, we have developed a very simple algorithm for feature detection in an indoor environment by means of a single camera, based on Canny Edge Detection and Hough Transform algorithms using OpenCV library, and proposed its integration with existing feature initialization technique for a complete Monocular SLAM implementation. Following this, we have developed hardware accelerators for Canny Edge Detection & Hough Transform and we have compared the timing performance of implementation in hardware (using FPGAs) with an implementation in software (using C++ and OpenCV).

## **Acknowledgements**

I would like to express my sincere gratitude towards Professor William Michalson for giving me opportunity to explore the exciting field of Monocular SLAM of which I was previously oblivious and for his invaluable support and directions while I was learning about concepts which were completely alien to me. I would also like to thank him for guiding me in my desire to integrate my area of specialization in FPGA design with Monocular SLAM. I thank my friends from Robotics Engineering: Supreeth Rao, Benzun Wisely and Vinayak Jagtap, for helping me understand Computer Vision concepts. I extend my thanks to my friend Mihir Vaidya for providing his insights in RTL design. Finally, I would like to thank my parents for providing me constant support from the other side of Earth.

# Contents

List of Figures .....	<b>VII</b>
<b>1. Introduction to SLAM.....</b>	<b>1</b>
1.1 Bearing-Only SLAM.....	4
1.2 Extended Kalman Filter .....	5
1.3 Inverse Depth Parameterization .....	8
1.4 Feature Initialization.....	10
1.5 Hough Transform based Feature Detection.....	17
<b>2. Feature Detection Algorithms .....</b>	<b>21</b>
2.1 Introduction to pillar detection.....	23
2.2 OpenCV Functions .....	23
2.3 Pillar detection with stationary Camera.....	28
2.4 Detection of Pillars with known shades .....	30
2.5 Pillar detection for Bearing-only SLAM.....	33
2.6 Detection of Floor Edges .....	47
2.7 Synchronization with delayed initialization and EKF update .....	51
2.8 Initial experiment in map building with one known feature.....	55
<b>3. Feature Detection Using FPGAs.....</b>	<b>65</b>
3.1 Hardware Acceleration Using FPGAs.....	65
3.2 Image Processing for Feature Detection on FPGAs.....	67

3.3	Canny Edge detection using FPGAs .....	68
3.4	Hough Transform using FPGAs .....	84
3.5	Timing Results.....	99
<b>4.</b>	<b>Conclusion .....</b>	<b>101</b>
	Bibliography .....	<b>104</b>

# List of Figures

Figure 1: Kalman Filter Equations [9] .....	7
Figure 2: Inverse Parameterization [5] .....	9
Figure 3: Creation of Parallax.....	11
Figure 4: Intersection of two bearing measurements [12] .....	12
Figure 5: geometric series of Gaussian distribution [A].....	14
Figure 6: Ray update on four consecutive poses [11].....	16
Figure 7: Projection of a line as a point on the plane for inverse parametrization [15].....	19
Figure 8: depth-range parameterization for floor line [15].....	20
Figure 9: Feature in Hallway characterized by vertical lines .....	22
Figure 10: Floor edges of Hallway characterized by inclined lines.....	22
Figure 11: A line with rho-theta parameters .....	25
Figure 12: False edge removal from Hough transform output .....	30
Figure 13: Pillar detection using a shade threshold .....	31
Figure 14: Pillar detection using a shade threshold and false edge removal .....	32
Figure 15: Two pillars far apart .....	34
Figure 16: A door frame and a Pillar far apart.....	35
Figure 17: Multiple pillars cluttered together .....	35
Figure 18: Possible path traced by a robotic platform in a hallway.....	36
Figure 19: Pillar detection algorithm for bearing-only SLAM.....	38

Figure 20: Pillar detection result1 .....	40
Figure 21: Pillar detection result 2.....	41
Figure 22: Pillar detection result 3.....	42
Figure 23: Pillar detection result 4.....	43
Figure 24: Pillar detection result 5.....	44
Figure 25: Pillar detection result 6.....	45
Figure 26: Pillar detection result 7.....	46
Figure 27: Inclined floor edge detection.....	47
Figure 28: Original scene that mimics a hallway.....	48
Figure 29: Floor edge detection .....	48
Figure 30: Corner feature.....	49
Figure 31: Floor edge detection algorithm.....	50
Figure 32: Delayed Initialization .....	52
Figure 33: Synchronization with EKF .....	53
Figure 34: Example case for synchronized feature detection and EKF update .....	54
Figure 35: distance-delta rho observation.....	57
Figure 36: distance calibration.....	58
Figure 37: lateral shift-leftmost rho observation .....	60
Figure 38: lateral shift calibration.....	61
Figure 39: sketch for map display.....	63
Figure 40: Initial feature position .....	64
Figure 41: display of first shift in camera with respect to known pillar .....	64
Figure 42: Image processing for Canny edge detection and Hough Transform.....	67



Figure 43: Smoothed edge [1].....	69
Figure 44: gradient of the curve [1] .....	70
Figure 45: edge location [24].....	70
Figure 46: Gaussian stage and the next stages.....	74
Figure 47: Inner hierarchy of next stages .....	74
Figure 48: Gaussian stage .....	75
Figure 49: Gradient calculation stage .....	76
Figure 50: Direction encoder .....	77
Figure 51: NMS and double thresholding stage .....	77
Figure 52: NMS calculation.....	78
Figure 53: Resource utilization for Canny edge detection.....	80
Figure 54: Input gray scale image.....	81
Figure 55: edge detected image .....	82
Figure 56: Edge detection result 1.1 .....	83
Figure 57: Edge detection result 1.2 .....	83
Figure 58: Edge detection result 1.3 .....	83
Figure 59: Mapping in parameter space [22].....	85
Figure 60: Complete Hough Transform Block Diagram [23] .....	88
Figure 61: Resource Utilization for generic Hough Transform.....	88
Figure 62: Inclined floor lines.....	89
Figure 63: detailed block diagram for Hough Transform.....	90
Figure 64: quadrant shifter block.....	91
Figure 65: calculation using DSP slices.....	92

Figure 66: Special unit of shift chain .....	94
Figure 67: Resource utilization for Customized Hough Transform .....	96
Figure 68: Original image before edge detection is performed .....	96
Figure 69: Edge detection performed on the original image .....	97
Figure 70: Input text file .....	98
Figure 71: Output waveform.....	98
Figure 72: timing performance of software implementation .....	99
Figure 73: timing performance of hardware implementation .....	100



# Chapter 1

## Introduction to SLAM

The SLAM concept is combination of two individual concepts: Localization and mapping. Localization is the deduction of the location of the autonomous vehicle if a map is previously known. Mapping, exactly opposite to the concept of localization means, with location known, developing a map with help of environmental features. The SLAM involves creating a map and locating the robot simultaneously, with neither the map nor the location known beforehand. One purpose of SLAM is to build a map of an unknown environment by a mobile robot while at the same time navigating the environment using that same map. It is for this reason that SLAM problem is very commonly referred to as chicken and the egg problem. SLAM works using probability models and makes use of the Extended Kalman filter which constantly updates the location of the robot relative to the features in the environment with best possible estimate of their location. The setting for the SLAM problem is that of a vehicle with a known kinematic model, starting at an unknown location, moving through an environment containing a population of features or landmarks [4].

The entire process of SLAM consists of multiple parts such as landmark extraction, data association, state estimation, state update and landmark update. The whole system is based upon a state vector and its iterative update. The state vector accounts for two factors: the original robot motion and the state of each feature that the robot detects in the environment. There is a continuous stacking of new features into the state vector as they are discovered, which is called feature initialization. This state vector is to be updated iteratively. The best estimate of the

update is provided by the Extended Kalman filter. The EKF maintains continuous tracking of an estimate of robot's probable position and also an estimate of uncertainty in the landmarks that it has seen in the environment.

The EKF is the heart of the SLAM process and its functionality is based upon two very important matrices; one of which is the mean state vector and another is a co-variance matrix. As mentioned earlier, the state vector defines the robot motion or robot position. Very simply we can say that it contains robot position in terms of  $x$ ,  $y$  and  $\theta$ . However its complexity may increase depending upon the situation. This will be described in detail in the next section. The state vector also contains another group of parameters that represent each detected feature. The most basic parameters for this would be the  $x$  and  $y$  coordinates of the feature, its elevation from the plane in which robot moves and the azimuth angle. The state vector represented in terms of the mean of its parameters forms the mean state vector. The second most important factor for the SLAM process would be the covariance matrix. The localization and mapping both can be mathematically described in terms of these two matrices. The co-variance matrix is nothing but a matrix containing the calculated individual co-variances of each parameter of state vector with each other parameter of the state vector. This state vector is generally represented as the single column matrix. Hence, a state vector containing 3 parameters (like  $x$ ,  $y$  and  $\theta$ ) will have a  $3 \times 3$  co-variance matrix. However, once the features are detected and their parameters are also added to the state vector, the elements of the state vector begin to grow, and consequently, the co-variance matrix increases in size accordingly. These two matrices are then updated every iteration using an Extended Kalman filter. The EKF is nothing but set of equations containing various matrices. It involves calculation of another parameter called as the Kalman gain and then based on it, calculates the estimated updates for the state vector matrix and co-variance matrix in

each iteration. Propagation of the full map covariance matrix is essential to the solution of the SLAM problem. It is the cross-correlations in this map covariance matrix which maintain knowledge of the relative relationships between landmark location estimates and which in turn underpin the exhibited convergence properties [4].

The SLAM process begins with map initialization. In any SLAM algorithm the number and location of landmarks is not known *a priori* [4]. Landmark locations must be initialized or inferred from observations alone. We detect a feature and initialize it, by adding it to the state vector. This is followed by the feature association. This is actually detecting whether the currently detected feature is same as an old feature or is a new one. After the map is initialized we begin the iterative steps. The whole iteration can be explained in the following steps: We predict the next state mean and co-variance matrices from the current state. Then, based on an observation model, we calculate the observed state values. The Kalman gain is calculated in this step and then, based on the updated gain, a new, corrected, state vector and co-variance matrix are calculated. In the third step, we add new feature parameters to the state vector, which adds an extra row and column to the co-variance matrix. The addition of new feature parameters is reserved for the third step because adding it in the second step would increase the required number of computations.

The mapping process ends, when a map closure (or loop closure) is achieved. Detecting closure is effectively recognizing a previously mapped path. It is similar to data association, but in this case the estimated uncertainty reduces to a great extent. Map closure generally occurs after a mapped path is traced back to its starting point.

Now, following this basic knowledge of the SLAM process, we can give an overview of our research and describe the structure of this thesis. The research pertaining to this theses

focused on developing efficient hardware implementations of algorithms for feature detection in an indoor environment, and for Monocular SLAM. This thesis does not intend to develop techniques for implementing a full-fledged SLAM, but it presents efficient algorithms for extracting features from the environment for the purpose of a monocular SLAM implementation. This chapter begins with basics of monocular SLAM and will go into the details of its three important components: the Extended Kalman Filter, Inverse Parameterization and Feature Initialization. We present work done in these specific areas of monocular SLAM. Finally, we explain line-based SLAM using Hough transform, which is our primary vision algorithm for feature detection. In the next chapter we describe our own feature detection algorithm and present its experimental results. We also go into the details of how our feature detection algorithm can be fused with feature initialization techniques of the first chapter to implement monocular SLAM. The last chapter deals with design and implementation of hardware accelerators for image processing, which can be used to execute our developed feature detection techniques in a more time-efficient manner.

## **1.1 Bearing-only SLAM**

When camera is used as a sensor rather than a sensor like a laser scanner, then we call it as visual SLAM [7]. In general, feature detection for visual SLAM can be done either by using stereo vision (two camera) or mono vision (single camera). Extensive work has been done in the field of range-bearing SLAM and range-only SLAM. However, the ranging sensors used by these systems (laser rangefinder, SONAR etc.) are unreliable, large and may be expensive. Using a single camera makes the sensor hardware on robotic platform inexpensive. This leads to the concept of monocular vision based SLAM called the bearing-only SLAM. Using a camera as a

sensor has added advantages, since they are well suited for embedded systems and work on low power. However, there is a huge drawback in using monocular vision since we lose a dimension (i.e. range or depth). Using single camera or monocular vision based SLAM is difficult to implement because as compared to stereo vision since monocular vision cannot comprehend a 3-D world directly. That is because a single camera captures a single image at a time which is obviously only 2-D. The parameter 'depth' that is the third axis of any object in the environment is difficult to determine in a single frame. All calculations for monocular SLAM are done over a period of time by capturing multiple frames and comparing each of them.

Further, the EKF requires Gaussian representations for all the variables that are involved in the map, which include the robot pose and the landmark locations. However, landmark locations cannot be known from a single observation and special initialization techniques and two observations of the same feature from at least two separate robot poses are required. Thus, we can see there are two aspects of this problem: one is being able to recognize landmarks in an environment and track it consistently over number of frames (to ensure it's the same landmark); the second is to estimate the landmark parameters from multiple frames (including depth) and initialize them in the map. The second problem is the most crucial task in the monocular SLAM and we have done extensive literature review on this aspect. We will describe various initialization techniques later in the chapter.

## **1.2 Extended Kalman Filter**

State space is the representation of a system in terms of a group of scalar parameters, defined by a state vector  $x = [x_1, \dots, x_n]^T$ . These parameters are called state variables. The true accuracy of the state vector is never known, there is always some amount of uncertainty



associated with it. Assuming that the uncertainty associated with it is Gaussian in nature, we can represent each scalar parameter with its mean and deviation (or variance). This would then generate a mean state vector and co-variance matrix.

A Kalman filter is nothing but set of equations that helps us obtain the best estimated update for the mean state vector and co-variance matrix. These equations are based on a linear transformation of the co-variance matrix. That is, when state vector  $x$ , is transformed to  $y$ , by a system described by linear functions, its co-variance matrix is also transformed by:  $P_y = FP_xF^T$ , where  $F$  is a linear function and  $P_x$  is the original co-variance matrix. However, this is only valid if the system is linear.

An Extended Kalman filter, or EKF, gives a solution for a non-linear system, hence it is used instead of the basic Kalman Filter. The EKF is a Kalman filter that is linearized around work point. That is, the EKF implements a Kalman filter for the system dynamics that result from the linearization of the original nonlinear filter dynamics around the previous state estimates [8]. This linearization is achieved by taking the Jacobian matrix, instead of the original system function while performing transformation of the co-variance matrix. Just as in the earlier case, let us assume  $F$  to be the system function that transforms state vector  $x$  to  $y$ . However, in this case  $F$  is non-linear. The Jacobian of  $F$  is its partial derivative with respect to the state vector  $x$ , about its mean. It is represented by, say  $F_J$ . Now, the co-variance matrix is transformed by:  $P_y = F_J P_x F_J^T$ .

The Kalman filter and The EKF are very similar in terms of their equations except for the change in equations for the co-variance transformation as explained above. But this has a greater implication: In a Kalman filter the transformation of the co-variance matrix and the mean matrix are independent and hence, these operations can be performed offline. However, in the EKF we

are taking Jacobian of the functions which are partial derivatives of the system with respect to state vector computed at its mean, for the co-variance computation; thus, in an EKF, the mean calculations and co-variance calculations are coupled and cannot be performed offline. Like a Kalman filter, the EKF is an iterative process containing two steps: predict and update. We will not describe the derivation of these equations, but we have listed the standard equations for the Kalman filter, and thus the EKF, in the following Figure1:

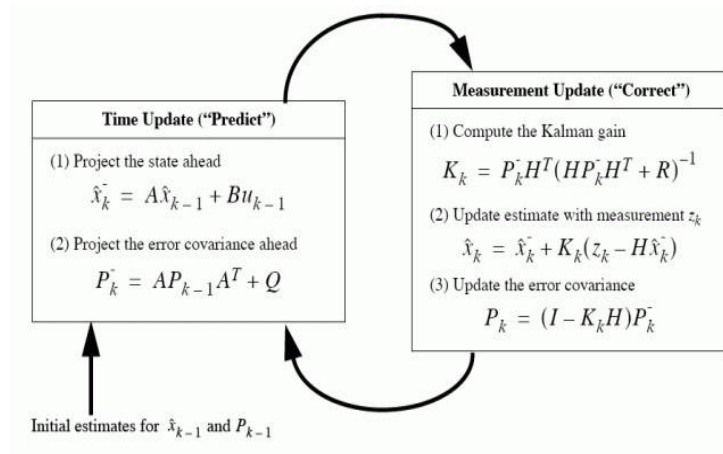


Figure 1: Kalman Filter Equations [9]

In applying these equations to the EKF,  $H^T$ , is the Jacobian matrix and  $K_k$  is the Kalman gain. In the equation to compute the Kalman gain, the part in parentheses is the observation model. It changes according to the available sensor for performing SLAM. We are performing the bearing-only SLAM, with a monocular camera. The observation model for monocular SLAM and the specific observation model that we will be using is described later in this chapter.

As we described earlier, calculating the Kalman gain and then updating the state vector and co-variance matrix form the second step of the SLAM iteration. The predict and update are a part of the Extended Kalman filtering operation, however in SLAM we also have to include the additional steps of feature association and the final steps of adding new feature parameters in the state equations.

### 1.3 Inverse Depth Parameterization

As compared to the generalized SLAM explained earlier, monocular SLAM will have some variations in its state vector. In the earlier section it was mentioned that the state vector is a combination of two groups of parameters: one is the motion model of the robot carrying the camera, the other is the group of parameters that represent the features of the environment. Let us first understand the motion model used in the state vector. We do not have odometry, hence we do not know the speed and distance, so we use motion models that estimate the linear and angular speed of the camera [5]. This is a constant velocity model with unknown acceleration inputs. These linear and angular accelerations are represented by zero mean Gaussian noise with known standard deviations ( $\sigma_a$  and  $\sigma_\alpha$ ) [6]. Equation (1) provides the equation of motion and its update equation for the camera.

$$f\mathbf{v} = \begin{pmatrix} r_{k+1}^{WC} \\ q_{k+1}^{WC} \\ v_{k+1}^{WC} \\ w_{k+1}^C \end{pmatrix} = \begin{pmatrix} r_k^{WC} + (v_k^W + V_k^W)\Delta t \\ q_k^{WC} \times q((w_k^C + \Omega^C)\Delta t) \\ v_k^W + V^W \\ w_k^C + \Omega^C \end{pmatrix} \quad (1)[6]$$

The camera's state vector  $f\mathbf{v}$  comprises a metric 3D position vector  $r^W$ , orientation quaternion  $q^{RW}$ , velocity vector  $v^W$ , and angular velocity vector  $\omega^R$  relative to a fixed world frame  $W$  and robot frame  $R$  carried by the camera<sup>1</sup> [7]. Now, this forms one section of the state vector. Another, section is formed by the parameters of the features of the environment that are calculated by technique called 'inverse depth parameterization'. In this technique, when a feature is newly initialized from a monocular camera, only information about the ray from the camera

---

<sup>1</sup> In case of Monocular SLAM, the frame of reference of the robotic platform can be represented as either robot frame 'R' or camera frame 'C'; both mean the same and is carried by the camera.

can be retrieved [6]. Equation (2) equation from [5] shows, the parameter  $y_i$  that represents the feature.

$$y_i = (x_i \ y_i \ z_i \ \theta_i \ \phi_i \ \rho_i) \quad (2)$$

In order to understand above equation we have to consider Figure 2:

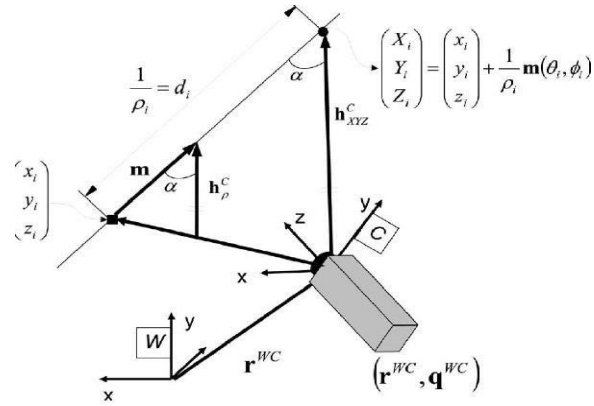


Figure 2: Inverse Parameterization [5]

As illustrated in Figure 2, the vector  $y_i$  encodes the ray from the first camera position from which the feature was observed by  $x_i, y_i, z_i$  (the camera optical center), and  $\theta_i, \phi_i$  (elevation and azimuth) defining unit directional vector  $m(\theta_i, \phi_i)$ . The point's depth along the ray  $d_i$  is encoded by its inverse  $\rho_i = 1/d_i$  [5]. In the Figure 2,  $\alpha$  is parallax and the line connecting the two camera positions is called the baseline. The baseline can be represented as a vector that represents translation motion of the camera. This baseline vector has been used in [13] as a criteria for feature initialization. The ray vectors created from two different camera positions are crucial to predicting the point location (feature position). Each ray vector can be expressed in the camera frame as

$$h^C = R^{CW} [X_i \ Y_i \ Z_i]^T \quad (3)$$

In equation (3),  $h^C$  is actually the camera observation through image projection. On applying the normalized retina model and hence the radial distortion model we get the feature position in terms of the camera center in pixels, focal length and pixel size.

These ray vectors, along with the baseline vector, can be used to calculate any of the angles (including the parallax) of the ‘triangle of vectors’ in Figure 2. Consequently, we can obtain the inverse depth (and hence depth) using the sine law. The 3-D point  $(X_i, Y_i, Z_i)$  can be represented in terms of its inverse depth as shown in equation (4):

$$x_i = \begin{bmatrix} X_i \\ Y_i \\ Z_i \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} + \frac{1}{\rho_i} m(\theta_i, \phi_i)$$

$$m = (\cos\phi_i \sin\theta_i, -\sin\phi_i, \cos\phi_i \cos\theta_i)^T \quad (4) [5]$$

Now, these two groups of parameters; the camera motion model and the matrix  $y_i$  calculated through inverse depth parameterization together form the state vector as in (5).

$$X = (x_y^T, y_1^T, y_2^T, \dots, y_n^T)^T \quad (5) [5]$$

## 1.4 Feature Initialization

Feature initialization is the tricky part in Monocular SLAM and as explained earlier, it requires at least two observations from two different robot poses (two different images) for the same feature or landmark. We know that, baseline and parallax, as shown in Figure 3, are the key to computing the depth of the feature. Generally, it is easier to create parallax for nearby features (as in the indoor environments) as compared to the objects that are far away since, when objects are close, even small camera translations are sufficient to create large parallax. In this sub-section we will go into details of the previous work done in initialization methodologies.

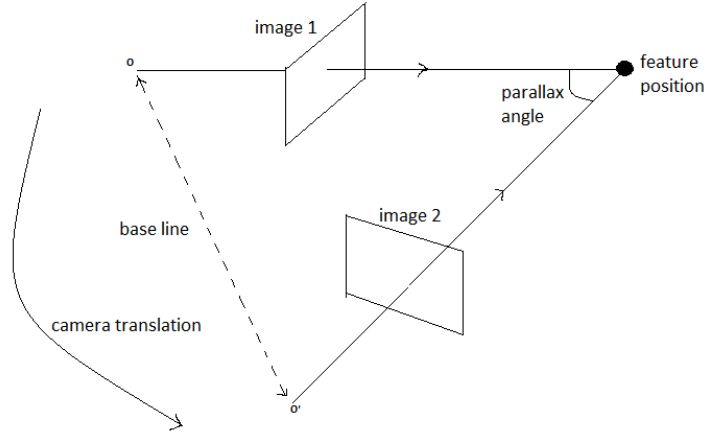


Figure 3: Creation of Parallax

References [13] and [14] describe the complete working of bearing-only SLAM based on concept of delayed initialization. Reference [13] uses a threshold for the parallax and the minimum baseline as a criterion for feature initialization. In [14] the initial PDF of a discovered landmark is approximated with a weighted sum of Gaussians and the initial state is de-correlated from a stochastic map, until it is confirmed as a landmark. The feature initialization is done by assuming *a priori* uniform distribution in the range  $[\rho_{min} \rho_{max}]$  for the depth  $\rho$  (assumed to be the surrounding environment for the robotic platform) and this *a priori* knowledge is approximated using the sum of Gaussians. Each Gaussian is then converted to Cartesian coordinates in the robot frame. Now, the likelihood of each Gaussian is computed based on further observations. Accordingly, bad hypotheses are pruned and when a single Gaussian remains, the feature can now be considered as a possible addition to the map.

Reference [12] presents a constrained initialization technique where the past poses of the robotic platform are retained in the SLAM state and feature initialization is deferred until their estimates become well-conditioned. In this subsection we will describe, under what conditions can the feature estimates be considered well-conditioned and how the feature will be initialized.

Consider the two bearing measurements and the vehicle poses as shown in Figure 4.

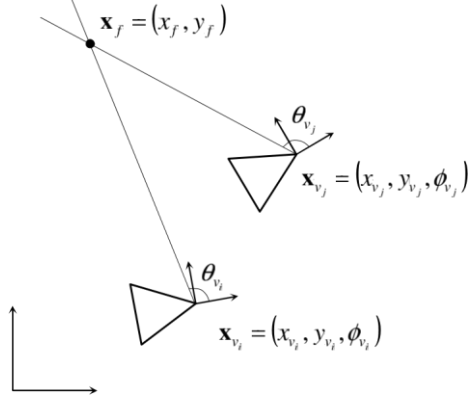


Figure 4: Intersection of two bearing measurements [12]

We know that a minimum of two bearing measurements are required to determine a landmark in using bearing-only SLAM. The location of the feature is calculated at the intersection of two lines. Now, according to [12] a pair of measurements can be considered well-conditioned if the true uncertainty distribution for feature location is well approximated by a Gaussian. If the pair of measurements is found to be well conditioned, the feature is added to the state vector. However, initially the feature location would obviously be ill-conditioned owing to various uncertainties in pose estimates, bearing measurements and the sufficiency of baseline created. The author's principle idea is this: the pdf of the new feature location should closely resemble Gaussian approximation obtained from a Jacobian-based linearized transform. This comparison between the joint pdf of the vehicle states, the initial Cartesian feature location, and the linearized Gaussian approximation is done through the calculation of sample relative entropy. Sample relative entropy is the statistic used to compare the two distributions: the true PDF of feature location and its Gaussian approximation obtained from Jacobian based linear transform. If  $f(x)$  is true distribution and  $g(x)$  is its Gaussian approximation then the relative entropy is defined as

$$D(f||g) = \int_{-\infty}^{\infty} f(x) \ln \frac{f(x)}{g(x)} dx \quad (6a)$$

However, the closed-form solution is not known for the above equation, hence, relative entropy cannot be calculated and instead *sample* relative entropy is calculated, which is obtained by sampling from true distribution  $f(x)$ . If  $f(x)$  is sampled as  $\{x^1, \dots, x^n\}$  then, approximate relative entropy is calculated as

$$D(f||g) \approx \frac{1}{n} \sum_{k=1}^n [\ln f(x^k) - \ln g(x^k)] \quad (6b)$$

The authors claim that computing sample relative entropy is an adequate measure to determine whether the measurements are well conditioned for feature initializations.

Now, we have seen that delayed and constrained feature initializations for the bearing-only SLAM problem can be effective, however these techniques require defining a criteria for deciding the sufficiency of the baseline and, consequently, there is a delay in landmark initialization until that criteria is fulfilled. As we know, the success of Monocular SLAM depends on the creation of parallax and a sufficient baseline. If the camera motion is too close to the direction of the landmark then, in order to create a sufficient baseline, the entire process of landmark initialization suffers a considerable delay. This is because, creation of acceptable baseline requires creation of adequately large parallax angle, and creation of such a parallax angle will take time if the direction of landmark is too close to the camera motion. The baseline sufficiency criteria requires vector calculations and calculations involving sine law to determine the generated baseline, and then binary decision has to be made depending on whether or not the generated baseline meets the pre-determined criteria and this makes it computationally expensive. Undelayed initialization is advantageous because it allows the use of landmarks that lie close to the direction of motion of the robot, for which the baseline would take too long to grow. This is crucial in outdoor navigation where straight trajectories are common and vision sensors will naturally look forward, resulting in baselines that grow very slowly [11].



The general mechanism for undelayed initialization is as follows: a set of hypotheses for the position of a particular landmark is included in the map. As the process moves forward, and more observations are made then, based on certain criteria bad hypotheses are purged and the ones with the highest possible likelihood are used to update the map. Reference [11] suggests a robust method for undelayed initialization. It uses an approximation of the Gaussian Sum Filter [GSF]. We will describe their method in brief to get a general idea. Consider the Figure 5:

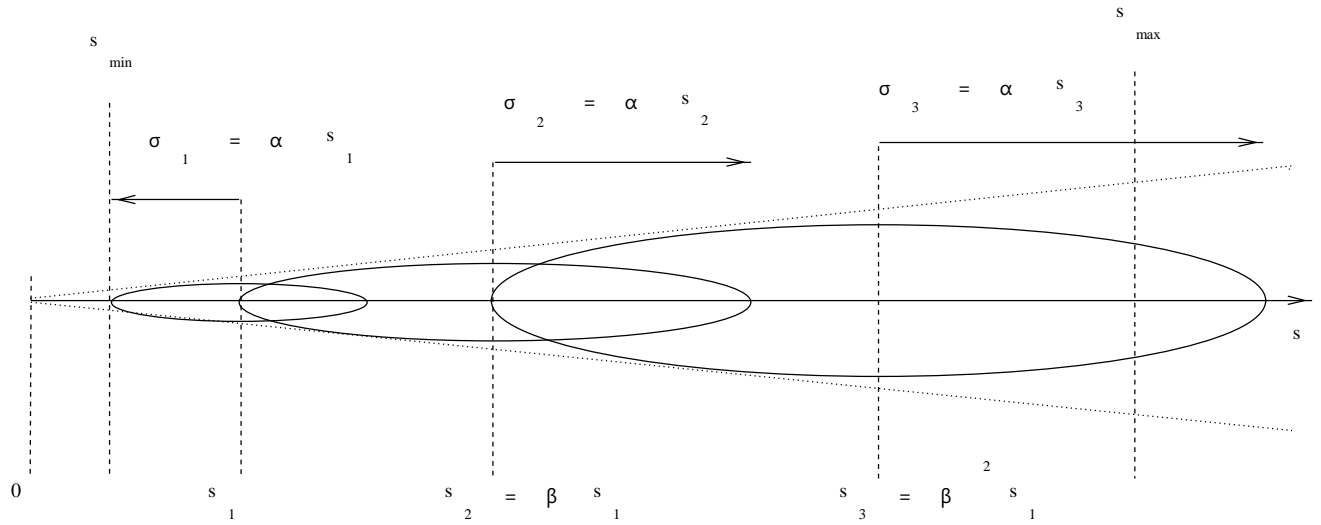


Figure 5: geometric series of Gaussian distribution [A]

When a landmark is first observed, the optical ray (along which that landmark is located) and its associated co-variances, define a conic probability distribution function as shown in the above diagram. The new landmark position that is to be stacked in the map, defined by  $x_p$ , is given by the function:  $X_p = g(X_v, b_p, s)$

Where  $b_p$  is the bearing information and  $s$  is the range (i.e. depth) which is unknown. Now, as shown in the above diagram, if  $s \in [S_{min}, S_{max}]$ , then  $p(s)$  is the Gaussian sum approximation defined as (7).

$$p(s) \approx \sum_{j=1}^{N_g} c_j \cdot \Gamma(s - s_j; \sigma_j^2)$$

$$\Gamma(s - s_j; \sigma_j^2) = \exp\left(-\frac{(s-s_j)^2}{2(\sigma_j)^2}\right) / \sqrt{2\pi\sigma_j} \quad (7)$$

Using this, we can create a hypothesis for a map with hypothetical landmark  $x_p^j$  at range  $s_j$ . However, this is only one hypothetical map. The pdf of the actual map state is the weighted sum of all the Gaussian maps given by (8).

$$p(X^+ | y_p) = \sum_{j=1}^{N_g} c'_j \cdot \Gamma(X^+ - X_j^{\hat{}}; P_j) \quad (8)$$

Now, we will have, say  $N_g$  maps for every landmark identified and, in case where  $m$  concurrent landmarks are identified and are to be initialized, we will have  $N_g^m$  maps. This multiplicative increase makes the solution untenable and hence, [11] proposes a minimal implementation of (7).

With reference to Figure 5 (refer page 14), assuming that the ratio  $\alpha_j = \sigma_j/s_j$  is small enough we take  $\alpha_j = \alpha = \text{constant}$ . Now, the minimal representation of (7) is:

$$p(s) = \sum_{j=1}^{N_g} c'_j \cdot \Gamma(s - \beta^{j-1}s_1, (\beta^{j-1}\sigma_1)^2) \quad (9)$$

In order to determine the first term ( $s_1, \sigma_1$ ) and the number of terms,  $N_g$ , the authors have imposed the conditions  $s_1 - \sigma_1 = s_{min}$  and  $s_{N_g} + \sigma_{N_g} \geq s_{max}$ . With the help of this minimal implementation, the conic shape ray along which the landmark lies can be filled with a minimum number of Gaussian shape distributions. Also, if each hypothesis is assumed to be a separate landmark, then all of them can be initialized in a single Gaussian map and then the standard EKF-SLAM procedure can be applied to it. Thus, the multiplicative growth we saw earlier can be avoided. The actual undelayed initialization takes place through simultaneous multiple stacking of all members of the conic ray each of which is assumed to be a feature. Finally, on

several iterations all but one members are purged based on the process described as follows (This eliminates the need for calculation of any initialization conditions and hence avoids delay):

All Gaussian shape members of the conic ray are stacked in the state vector assuming them to be different landmarks. Each hypothesis is stacked one by one iteratively by a standard EKF procedure, with each hypothesis having equal weight. This is depicted by a uniform Aggregated Likelihood vector as follows:

$$\Lambda = [\Lambda_1 \dots \Lambda_{N_g}]$$

$$\Lambda_j = 1/N_g \tag{10}$$

The fully correlated map is then updated by federated information sharing. This works on the principle described in detail in [11]. Finally each hypothesis is successively updated with its measure of likelihood (11).

$$\Lambda^+_j = \Lambda_j \cdot \lambda_j. \tag{11}$$

The Aggregated Likelihood (AL) vector  $\Lambda$  is then normalized so that  $\sum_j \Lambda_j = 1$ . A simple threshold on the AL which is dependent on the actual number  $N$ , of remaining members. Ray member  $j$  is deleted if  $\Lambda_j < \tau/N$  where  $\tau$  is in the range [0.0001 0.01] [11]. When  $N = 1$ , the ray has collapsed to a single Gaussian and now just a standard EKF SLAM. The diagram 6 represents the undelayed initialization process:

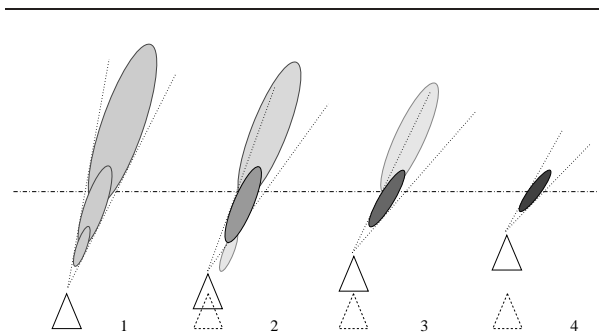


Figure 6: Ray update on four consecutive poses [11]

## 1.5 Hough Transform based feature detection

Performing Hough transform on an input image forms an essential component of this thesis. In our implementation of Monocular SLAM, we will be using Canny edge detection followed by Hough transform, to detect the landmarks and extract their parameters. We will be performing Hough transform on an input image in software followed by an hardware FPGA implementation and will analyze performance in each case. However, it is important to understand the role Hough transform plays in our SLAM implementation. In this section we will explain, how Hough transform based extraction of lines can be used to implement monocular SLAM.

Feature detection is a primary step in a SLAM implementation. In order to build a map of the environment, it is essential to be able to detect certain distinct landmarks and features that characterize that particular environment. The next step in SLAM is to be able to extract parameters of the detected landmark. It is very common to use Harris corner detection for this purpose. In this thesis, we have made effective use of Hough transform to detect features and extract their parameters in an indoor environment. Most of the SLAM implementations for indoor environments are based on corner point detection like in [10]. Although not very common, Hough transform has been used previously for SLAM implementation, especially in detecting indoor environments such as a hallway. This is obviously because the structural properties of a hallway or a corridor are explicitly based on vertical and horizontal lines. One can easily characterize the floor, pillars and beams, door edges etc. in terms of parallel lines of particular orientation in a 2-D plane. In [15] the authors have presented the lines representing floor edges and vertical edges as sensory inputs for the monocular SLAM, something very similar to what we have tried to implement. In [16] they have used Hough transform for SLAM in a very

specific type of indoor environment. They have considered a plane environment having lines on the floor such as those in many universities, malls, museums, hospitals, houses, and airports. Their idea is to make effective use of Homography. The lines on the floor are identified by the Hough transform, and the parameters of the normal representation of the straight line returned by the Hough transform are mapped to the world using a pre-calculated homography matrix, which are then plugged into updating phase of the EKF. However, this kind of implementation would be highly non-generic. Another related work can be found in [17] which presents a mapping algorithm for bearing only sensors, based on the Fast Hough Transform. In [18] a geometrically constrained EKF framework for a line feature based SLAM has been implemented, which is applicable to a rectangular indoor environment. It makes use of a constrained Hough voting space, which they have developed in accordance with their constrained EKF framework mentioned earlier, as opposed to a conventional voting space. Reference [19] describes in detail a scan matching algorithm using Hough transform, which can be used for SLAM implementation.

In our thesis we depend heavily on the technique used by [15] to implement feature detection algorithm for bearing-only SLAM. As we will explain in the later chapters, the feature that we try detect in the environment is a pillar (or all pillar-like features) and the floor edges, which are characterized by a straight line. Hence, the observation model used by [15] is effective. We will go through their line-based SLAM technique in brief.

A detected line is projected to the ground as a point. In our case it will be a pillar which will be projected on the ground plane and that projection would determine the pillar position with respect to the moving robotic platform. The point on the 2-D plane is as shown in Figure 7.

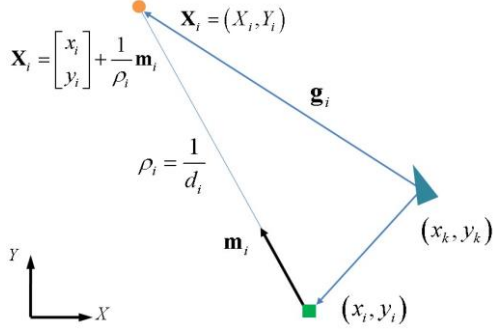


Figure 7: Projection of a line as a point on the plane for inverse parametrization [15]

We have seen, in Section 1.3, the inverse parameterization technique for a 3-D point in space however, in this case, it is a point which lies on the same plane and hence, it does not have the elevation angle. The inverse parameter equations (4) are adjusted for this point as (12):

$$X^i = \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \frac{1}{\rho_i} m_i$$

$$m^i = \begin{bmatrix} \cos \alpha_i \\ \sin \alpha_i \end{bmatrix} \quad (12)$$

Now the observation model, similar to equation (3), adjusted for the projected point is as (13):

$$g_i^c = R^{CW} \left( X_i - \begin{bmatrix} x_k \\ y_k \end{bmatrix} \right)$$

$$h_i = c_x - f_x \frac{g_y}{g_x} \quad (13)[15]$$

As seen earlier, the projected feature point is then coded by its inverse depth as  $y_i = (x_k, y_k, \alpha_k, \rho_k)$ . However, unlike equation (2), there is no elevation angle. Rather, the angle  $\alpha_k$  defines an angle made by the observation ray with X-axis. Thus,  $x_k, y_k$  and  $\alpha_k$ , describes the robot pose and  $\rho_k = 1/d_k$  describes the inverse depth.

Another feature of interest to us is a floor edge. In [15] they have encoded the floor edge with its depth (instead of inverse depth) as  $y_i = (x_k, y_k, \alpha_k, d_k)$ . The observation model for edge, as given in equation (14) is:

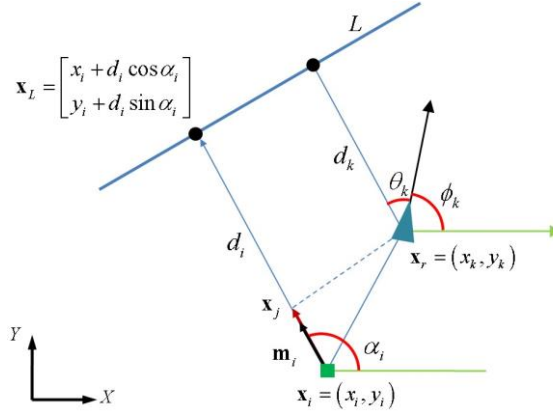


Figure 8: depth-range parameterization for floor line [15]

$$\overline{x_i x_j} = m_i (m_i \cdot \overline{x_i x_r})$$

$$\overline{x_j x_L} = \overline{x_i x_L} - \overline{x_i x_j} = \begin{bmatrix} x_{jL} \\ y_{jL} \end{bmatrix}$$

$$h_i = \begin{bmatrix} \theta_i \\ d_i \end{bmatrix} = \begin{bmatrix} \alpha_i - \phi_k \\ \sqrt{x_{jL}^2 + y_{jL}^2} \end{bmatrix} \quad (14)[15]$$

## Chapter 2

# Feature Detection Algorithms

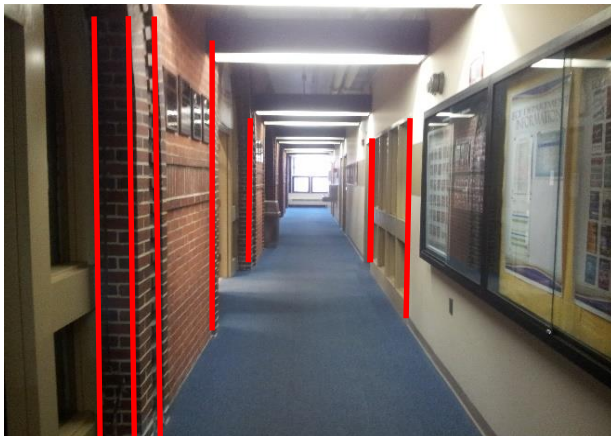
We have seen various feature initialization methods for bearing-only SLAM in Chapter 1. In a SLAM implementation, the task that precedes feature initialization is feature detection. Only once the identifiable feature is confirmed as detected, we can begin with the initialization process. Harris corner detection is the most commonly used image processing algorithm for the purpose of feature detection in SLAM. The 3-D corner point detected can be coded using its inverse distance as we have seen in Section 1.4. However, for our implementation, we will draw heavily from the concept of line based SLAM described in Section 1.5, and we will be using Hough Transform as our primary vision algorithm.

The initial goal of our research was to design an efficient algorithm for feature detection in an indoor environment for the purpose of Monocular SLAM. An ideal indoor environment would be a hallway. Generally, Hallways are characterized by several features with vertical edges like beams, pillars, doors etc., as in Figure 9; also, a hallway can be traced by its parallel floor lines, as in Figure 10. The Hough transform is a line detection algorithm, and hence is perfect for such conditions. Also, as described in Section 1.5, a detected pillar can be projected to the floor as point on the floor plane and can be coded using inverse parameter as described in [15].

In this chapter we will present our pillar and floor detection algorithm and its variations. We will also present our results for feature detection experiments. Further we will talk about,



how we can implement a complete bearing-only SLAM system using our feature detection algorithm.



*Figure 9:*

The hallway is characterized by features that can be represented by parallel vertical lines. These features can be described as being pillar-like and can be detected using Hough Transform.

*Figure 10:*

The floor edges of the hallway are characterized by inclined lines. The Hough transform algorithm can effectively detect lines at specific orientation such as these yellow lines.

## 2.1 Introduction to Pillar Detection

A pillar is easily identifiable because it is characterized by the two parallel lines which form its edges. In this thesis, we use “pillar” as a generic term for any pillar-like features like doors, beams, protruding wall edges, window frames, etc. Basically, any feature of acceptable width which can be identified by its two parallel edges, is considered a pillar in our feature detection implementation. The two parallel edges are extracted by a combination of two image processing algorithms: Canny edge detection and the Hough Transform. We will go into the intricate details for each algorithm later in the thesis, for now, it is sufficient to know, that Hough transform detects lines from a Canny edge detected image and returns its coordinates in the parameter space (i.e. rho and theta). This information extracted by the Hough transform algorithm can be used for computing virtual widths of the detected possible pillars and will be instrumental in confirming the existence of a pillar and tracking it over successive frames.

When a robotic platform moves in an indoor environment it has to map all possible identifiable features (in our case all pillar-like features). However, as we move towards designing an algorithm to do so, first we need to develop a basic code that can identify a stationary pillar with a stationary camera. We have developed the code in C++ and we have used OpenCV functions. Hence, it is necessary to go into a little detail of these functions and OpenCV in general.

## 2.2 OpenCV Functions

Our objective is to design a SLAM application using no other sensor or data acquisition sensor except the monocular camera. The feature detection and calculation of parameters for the Kalman filter will be based upon a series of images captured over a period of time. In order to extract data from a captured image, we will require understanding of various image processing

techniques. OpenCV is going to be of utmost importance in this case. The basic description and key points regarding OpenCV are described in [2]. OpenCV is an open source computer vision library which is written in C/C++ and runs under Linux, Windows and Mac OS X. OpenCV was designed for computational efficiency, with a strong focus on real-time applications. It is written in optimized C which can take advantage of multicore processors and it is intended to provide a simple-to-use computer vision infrastructure that helps people build fairly sophisticated vision applications quickly. The OpenCV library contains over 500 functions that span many areas in computer vision. In our code we will be making extensive use of OpenCV functions which enable us to perform various image processing task without the need for understanding the intrinsic details of image processing algorithms. Out of all the functions tried and tested, the functions that have been finally used in the code are listed in this section.

#### Canny edge detection:

The basic concept of edge detection is based upon the fact that an edge is characterized by a sudden change in intensity between two consecutive locations (in same orientation) in the image. Mathematically, amount of change in intensity with respect to spatial distance (called the first derivative) is the basis to create edge detection operators. While checking for the edges in the image on multidimensional basis, we take partial derivative along each axis each of which is called a gradient. The edge operators function on the principle of approximating the local gradient of the image function. The Canny function also, at core, works on this gradient principle when used in basic (single scale) form [1]. The Canny operator is considered to be superior to other edge detection algorithms in its single scale form [1]. In our application we will be using the OpenCV function for Canny edge detection. The function is of the form [3]. We will be using this function before performing a Hough transform on the image.

```
Void Canny(InputArray image, OutputArray edges, double threshold1, double threshold2, int apertureSize=3, bool L2gradient=false )
```

### Hough transform:

Once edge detection is performed on an image, the Hough transform function looks for a large set of consecutive points along the same orientation and assumes that to be a line. It then calculates the perpendicular distance from an imaginary origin to that point and the inclination of that perpendicular line (angle with respect to the imaginary x-axis). Basically, it returns the details of all lines it can find in the image and returns the polar coordinates of that line as the values rho and theta. These parameters will be used extensively in our code for extracting various feature parameters. Theta is measured in radians<sup>2</sup>. The Hough transform detects all possible lines in the edge detected image. In our code we detect lines having a specific range of orientations and eliminate all other lines. Hence, we use conditions for lines having a theta value within a desired range of radians. We will detect vertical lines measuring approximately 0 radians (0 degrees) for the detection of pillar and lines of approximately 0.78 radians (45 degrees) for the detection of the floor edges. The rough sketch shown in Figure 11 describes the measurements for the Hough transform:

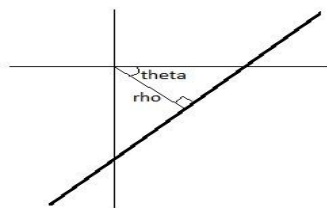


Figure 11: A line with rho-theta parameters

---

<sup>2</sup> The units of  $\rho$  is the units of the graph system in which the line is plotted. The resolution of  $\rho$  is 1 pixel in OpenCV. For our implementation on FPGA,  $\rho$ - $\theta$  space is an abstract mathematical construct where  $\rho$  value ranges from  $-n/\sqrt{2}$  to  $+n/\sqrt{2}$  for  $n \times n$  sized image

The OpenCV format for the Hough transform function is [3]:

```
Void HoughLines(InputArray image, OutputArray lines, double rho, double theta, int threshold, double srn=0, double stn=0 )
```

In our code, departing from the general convention of using feature key point detection as the main means of detecting a feature in the environment, we have used Hough transform as the primary feature detection tool. However, it has been found that Hough transform is sensitive and prone to error and as will be explained later, has a good detection range of only around 70-80 inches (177.8 cm – 203.2 cm). In future work we will may reconsider the use of feature key point detection, however, as of now we will continue to rely on the Hough transform method.

#### Line function:

The line function is a basic line drawing tool provided by OpenCV. For display of lines detected by the Hough transform we will require this function. The Hough transform does not display lines, it just detects them and returns the polar coordinates. Hence, in order to display the lines in a separate window to check its correctness, we have to call the line function. The line function requires two points with their respective x and y coordinates. The Hough transform however returns the polar coordinates. Hence, always after Hough transform is called, before we call the line function, we have to write a small routine to convert the polar coordinates of each line detected into the Cartesian coordinates to enable the line function to draw those lines. Now, another function called lines.Size() will be required in this routine. This function returns the number of lines that are detected by the Hough transform in that particular image. It is at this step that we can use our angle filter and eliminate the remaining lines while retaining those at our desired orientation. The line function will also be used by us in displaying the path traced by the camera in the two dimensional map. The OpenCV syntax for the line function is as follows [3]:

```
Void line(Mat& img, Point pt1, Point pt2, const Scalar& color, int thickness=1, int lineType=8, int shift=0)
```

### Circle function:

Similar to the line function, this is another basic drawing function of OpenCV. We will be using this function to draw the points of very low radiuses to display a point that indicates the location of the feature. This requires  $x$  and  $y$  coordinate of a point and the value of the radius to draw the circle. The OpenCV syntax of this function is [3]:

```
Void circle(Mat& img, Point center, int radius, const Scalar& color, int thickness=1, int lineType=8, int shift=0)
```

Besides these important functions of OpenCV, we will be using some basic functions such as *imshow*, which displays an image in the window titled by the name passed within the colons, in the function. We will be using the standard functions for capturing the image and storing the image or other functions that retrieve the image from memory which are not mentioned in detail here.

It is important to note that the  $x$  and  $y$  coordinates of the points which we will supply to all the functions are denoted in our code in the form of an object of the class *point*. If *pt* is an object of the class *point*, then *pt.x* and *pt.y* values determine the  $x$  and  $y$  axis of a particular point. Also, all images are represented in form of a matrix defined by the data type *Mat*. The variables of this data type are passed to the functions such as *imshow*. Also, all the image processing tasks are performed on the variables of this data type. Basic operations such as addition can be formed on them (and we will be performing on them for the two dimensional map display).

## 2.3 Pillar Detection with Stationary Camera

The map will be built based upon a set of features and the position of the camera with respect to the feature locations. In an environment like a hallway or a corridor pillars are the most easily identifiable features and hence, we begin with code that identifies a pillar. Our ultimate aim is to find the position of the camera with respect to the location of the pillar. This will be our first step towards being able to create a full-fledged map.

The pillar is easy to identify because it is characterized by two parallel lines which constitute its edges. We have seen the advantages of using the Hough transform for accurate edge detection and we can put it to use in our case. The Hough transform function returns the value of rho and theta of each line. If the lines are parallel, the theta for both the lines has to be same. In such a case, the difference between the rho values of both the lines would be the width of the pillar in the image.

However, the Hough transform will detect, not only the edges of the pillar but also all the possible edges in the image. That would include all the vertical, horizontal and the edges oriented at an angle. The pair of parallel lines detected in such a case would not necessarily mean a pillar. It could be anything present in the environment. Thus, a series of operations would have to be done on the image in order to filter the unnecessary data and detect the pillar.

Now, we begin with performing Canny edge detection over the captured image followed by the Hough transform. The necessary angle adjustments must be done in the code such that out of all the lines detected by the Hough transform, only, the lines which are vertical or close to being vertical will be retained and rest will be eliminated. This would provide us with the pillar edges that we require. This adjustment would form a crucial part of the code and would be

critical when we are simultaneously detecting the floor edges also along with the pillar edges in the later part. The angle adjustment condition used in the code is as follows:

$$(theta > CV\_PI / 180 * 170 \parallel theta < CV\_PI / 180 * 10)$$

This condition is based upon the theta values that Hough transform function recognizes as explained in the earlier section.

The final image with the detected pillar edges would, however, not be devoid of noise. Canny edge detection is based upon the pixel intensities. Thus, the edges or the features of the detected objects would not be sharp, obviously because the intensities are not uniformly distributed over the objects and depend on the light or shadow falling on them. The final image would end up detecting number of false edges. It is necessary to develop a logic that would eliminate the false edges and retain the required ones.

The algorithm we have devised to eliminate noise works in the following way: The number of lines detected by the Hough transform is returned by the function *lines.Size()*. We create two arrays that stores all the rho and theta values that Hough transform detects. Then using the *lines.Size()* function we parse through these arrays. If the theta of the two lines is same (indicating that they are parallel) we calculate the difference in their rho values (The difference in the rho values of the parallel lines representing the edges of the pillar is actually the abstract image width of the pillar) and store it in a third array. This would eliminate all the non-parallel false edges as shown in Figure 12. Third array now contains all possible image widths. Now, it is obvious that distance between an actual edge and the false edge will always be less than the distance between two actual edges. Thus, we take the largest value from the third array and that



would be the image width of the pillar that we call *delta rho* ( $\Delta\rho$ ). The parallel lines with a valid value of  $\Delta\rho$  would indicate the detection of the pillar.

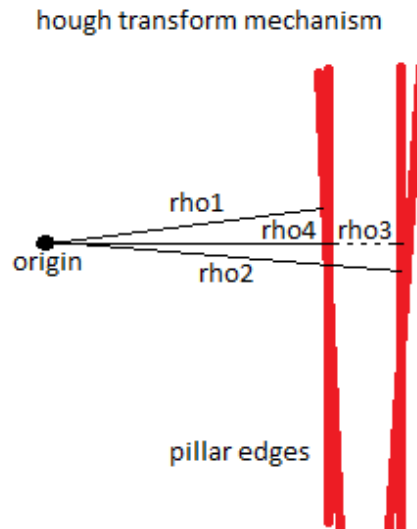


Figure 12: False edge removal from Hough transform output

## 2.4 Detection of pillars with known shades

As part of initial trials, we had used bright colored pillars. We have been able to successfully do the error free and accurate pillar detection for these pillars with known shades. Its error-free nature is obvious, because we apply threshold to the image and remove all the other colored objects from the image, which do not lie within are color intensity range that we pre-decide. This detection range is adjustable. In this section we have given details of this technique and presented the experimental results. The question would be, however, is detection of pillars with known shades of any use in real-world applications? In general, SLAM is used to map an unknown environment, however, some detail of the environment may always be available. We may know, whether it is an indoor or an outdoor environment. This would change the parallax requirement as explained in Section 1.4. If it is an indoor environment we must have some basic knowledge, like the presence of hallways (as in our case). If the environment is partially known,

for example a robot which performs tasks such as intra-office package delivery may have the knowledge of the kind of pillar shades present in the office space. In such cases this technique may be useful. However, moving forward with our thesis we will present an efficient generic algorithm for any pillar of unknown shade and width.

For now, since we are beginning with the basic steps, we will rely on the assumption that the pillar we wish to detect has a bright color such as red or orange. We will work with two pillars of different widths one red in color and other orange in color. Thus, our first stage of filtration would be detecting all the objects present in the environment that would have the intensity values corresponding red or orange colors. This, would eliminate most of the unnecessary objects in the image. When we implement object detection using its color, the resultant image would be a threshold image. In the Figure 13 we have a red pillar. The next image is the threshold image. Now, we can see that the threshold image does not give a sharp detection of the red pillar. Thus a number of false edges would be detected.

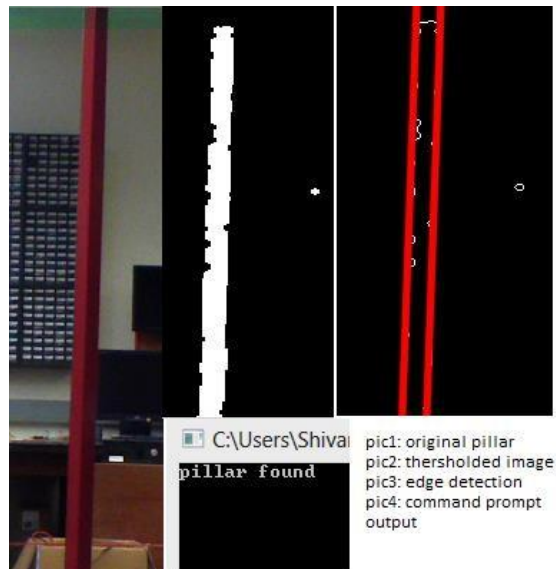


Figure 13: Pillar detection using a shade threshold

Once we eliminate the noise and calculate a valid  $\Delta\rho$ , our function successfully returns affirmative message detecting the pillar. In order to understand the intricate details of the logic, we will see the internal values of the arrays that we have used in our code and also see the process of noise elimination (removal of false edge as in Section 2.3 and Figure 12) and successful detection of the orange pillar with its width in the image (see Figure 14).

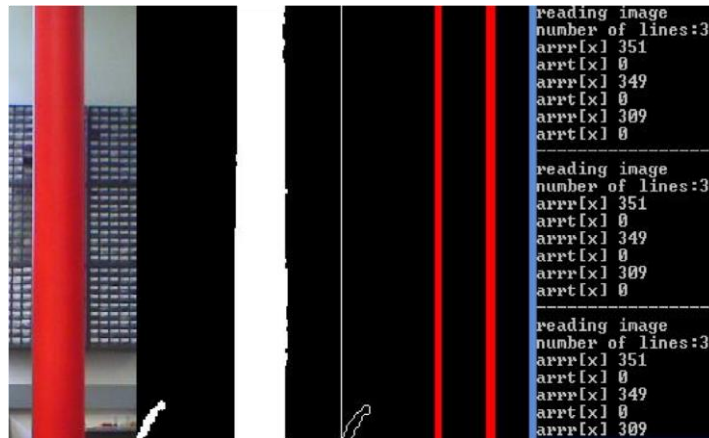


Figure 14: Pillar detection using a shade threshold and false edge removal

We have two arrays: `arrr[]` and `arrt[]` which store the rho and theta values respectively of each line. Now in the above image we see that the threshold image of the orange pillar is not sharp. The right edge contains certain distortions. Hence, the Hough transform will detect three edges (one of them being false). In the image we can clearly see that the red colored line on the right edge is thicker than that on the left edge. This is because it detects three lines instead of two. This can be seen on the command prompt output. The number of lines being detected is 3. The two arrays contain three theta and rho values of each line. We see that theta values of all three line is same i.e. 0. Which means all three are parallel. However, looking at their theta values we see that rho values of two edges are very close to each other (i.e. 349 and 351) while the third edge has rho value 309. In this case we can have three values of delta rho:

$$351-309=42$$

349-309=40

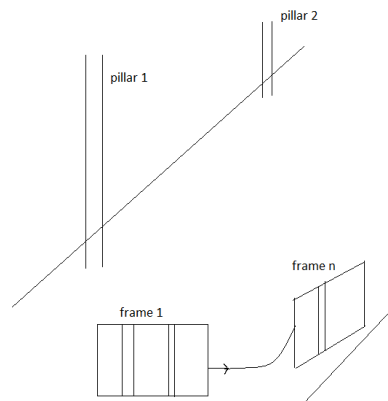
351-349=2

The highest of these three values that is 42, is taken as the valid value of delta rho and hence, the pillar detection is successful.

## **2.5 Pillar Detection for Bearing-only SLAM**

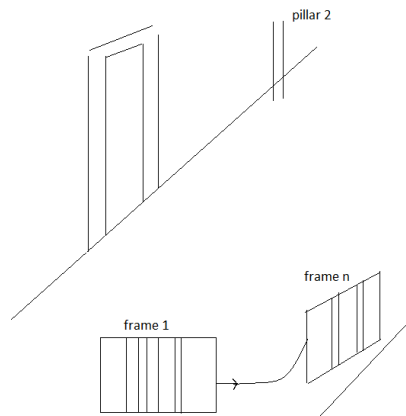
The pillar detection methods discussed in the earlier sections are highly non-generic. We began with detecting a single pillar with a stationary camera and then we went ahead to accurately detect pillars with known color shades. The code can be adjusted to include various intensity levels and color combinations and can be used if the environment conditions are previously known. However, this adjustment cannot be done in real-time obviously and it is highly unlikely that we will know the environment conditions previously in real world applications. Also, we cannot directly detect a pillar by identifying two parallel lines because, in general, there will be multiple pillars in the environment and thus, the image captured will detect multiple parallel lines. In this situation, it is impractical to detect a pillar simply by selecting largest virtual width, as we have done in Section 2.4. It is also important to note that unlike the stationary camera in the earlier case, we have a moving robotic platform with a mounted camera and hence, the image is prone to be noisier. This is where we need a well-defined algorithm to detect a pillar. Also, there other necessities which will be required by any bearing-only SLAM implementation that is, motion of camera to generate sufficient base-line and parallax for inverse depth parameterization. The pillar detection algorithm should conform to all such necessities.

Before designing the algorithm we consider various possible cases that we may encounter while moving a camera through the hallway. Consider the sketch shown in Figure 15. In the sketch we see that frame 1 captures multiple lines from both the pillars that it can observe. When we perform camera translation to create a parallax, the second pillar goes out of view. Now edges of only one pillar remains in the frame and we can use the same detection techniques that we described in Section 2.3. However, there is another issue here: there is no way of telling which pillar we are actually tracking, because depending upon the angle of the camera, the pillar remaining in the frame can be either of the two pillars.



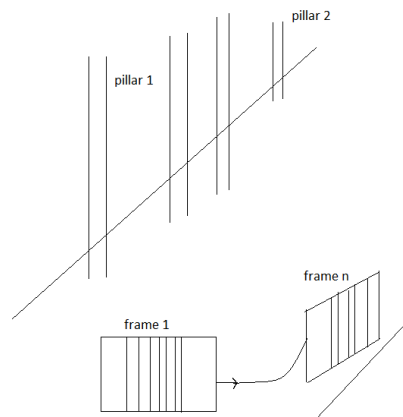
*Figure 15: Two pillars far apart*

Consider another possibility (Figure 16), there is a door closer to the camera and a pillar somewhere in the rear. Now, the door is characterized by two parallel pillar-like frames. Besides having the same issues as those in the earlier case, in this case we have the additional problem that two parallel frames of the door will never go out of view and there is no way of knowing whether it is a door or two parallel pillars.



*Figure 16: A door frame and a Pillar far apart*

In the next case (Figure 17) there too many pillars in the environment and it is difficult to extract one of them for feature initialization because some of them will always remain in the frame.



*Figure 17: Multiple pillars cluttered together*

An ideal case would is shown in Figure 18. The robotic platform would be able to isolate and extract each pillar with subsequent camera translation and be able to map the whole environment, but this will not always be the case, so our pillar detection algorithm must be able to cover all the earlier described cases.



This will account for all the noise in pillar detection. Two arrays are then created (or a single 2-D array) to store rho and theta for each line. All parallel lines (identified by same theta) are then taken together. This is performed by double loop iterations within the code. The difference in rho of each pair parallel lines defines the virtual width of a possible pillar. All the possible pillars with their virtual widths are then enumerated. Most of the tasks described above are similar to what we have done in the earlier sections. However, it is important to note few things: lines with same theta in earlier section arose out of presence of noise, however, in this case it may be due to presence of other pillar-like features within the environment (as we have seen in the sketches) or, as in the earlier case, may be due to presence of noise. We have no way of distinguishing them. Here we propose a two-fold procedure: the first is to simply apply threshold, that is, all virtual widths above a particular upper threshold and those below a particular lower threshold are eliminated. This would remove some of the very large widths which cannot possibly be pillars and would also eliminate noise caused due to false widths.

This however, would be a very crude method and not very accurate. It would eliminate widths of interest and end up retaining some large unviable widths which lie near the threshold margin. Now, once we have a list of all possible pillars, we have to select one of them as our current feature. Obviously we will select the pillar nearest to the camera, the ones that are away will eventually get detected as the robot moves towards them. The pillar with largest virtual width will be assumed to be the nearest. It is this hypothesis, although logical, will encounter several problems. As we have shown in the sketches earlier, there is no way of proving that there is a single pillar. It may be multiple pillars, or a doorway or edges of wall (although edges of the wall may be eliminated while applying a threshold).



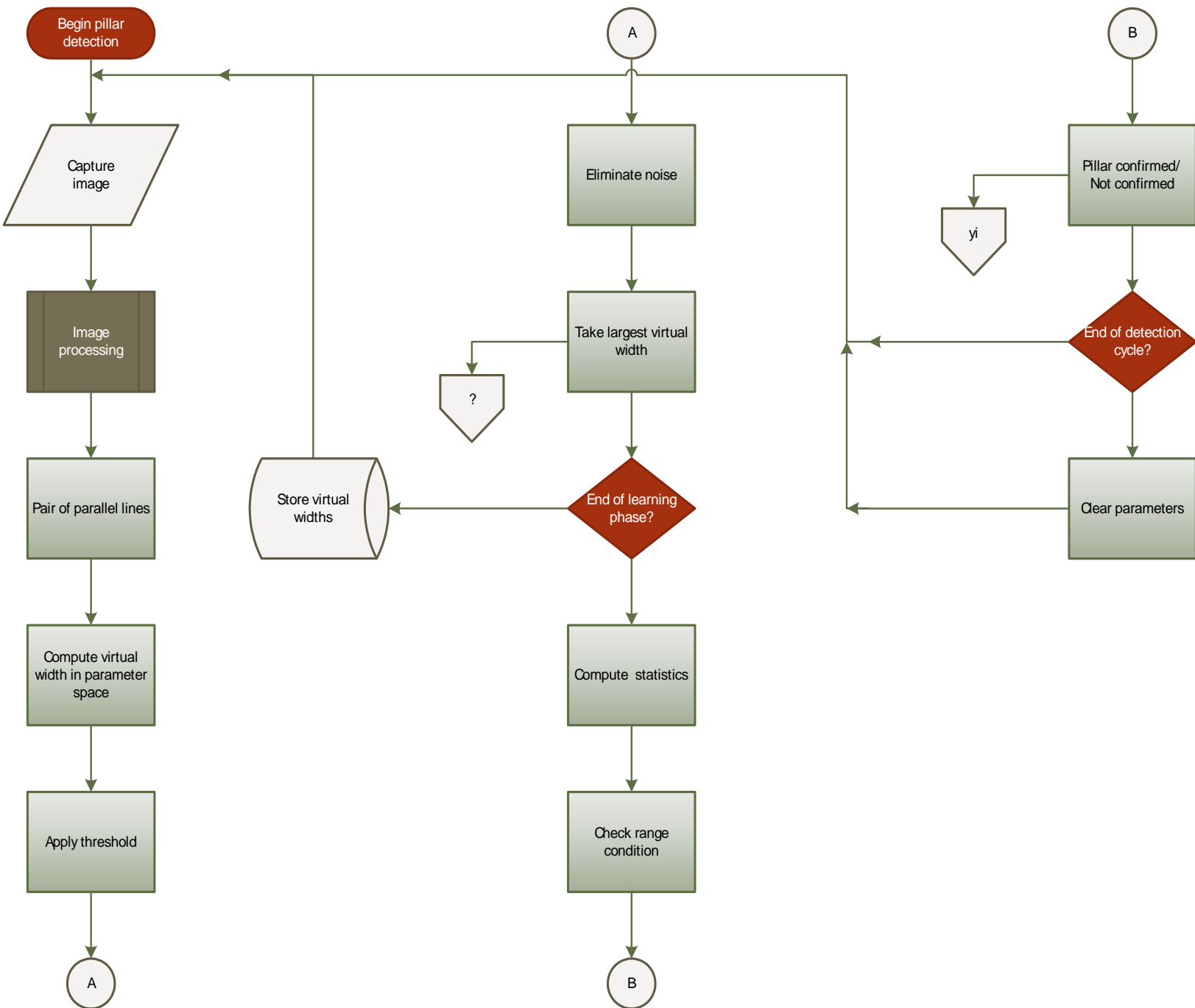


Figure 19: Pillar detection algorithm for bearing-only SLAM

This is where the second part of the algorithm comes into the picture. The second part of the procedure is to implement a ‘learning-checking’ routine. This is a very basic routine; it divides the detection cycle into two parts: a learning phase and a checking phase. The division of the total number of iterations of the detection cycle between the learning phase and checking

phase has to be carefully decided beforehand. During the learning phase, the virtual widths of each possible pillar are stored and, at the end of the phase, the average is calculated. A suitable range can be set deviating from this average. In the next phase, which is the checking phase, each detected virtual width is checked to determine if it lies within this range. This check is performed for all the image frames that are captured in the checking phase. If the number of hits is greater than number of misses, the pillar is confirmed as detected. Remember, during the entire detection cycle, the camera is continuously moving, the deviation range selected is extremely important, because if the camera moves too fast too close to the pillar, its virtual width may fall out of range even though it is the same pillar that we have been tracking throughout the cycle. Hence, all three parameters – detection cycle iterations, number of frames in the learning phase and number of frames in the checking frames – have to be decided according to the velocity of the robotic platform. Once, a pillar is confirmed, both its edges (two parallel lines) are projected to the ground plane as a point and coded using the inverse parameterization for a straight line as described in Section 1.5.

We have experimentally observed the working of our algorithm. In each case we have manually moved the platform towards the scene. Our detection cycle is 10 iterations long divided 5 and 5 between the learning and checking phases. The number of possible pillars detected in each iteration is printed and if the pillar is confirmed it prints so, otherwise it moves to the next detection phase.

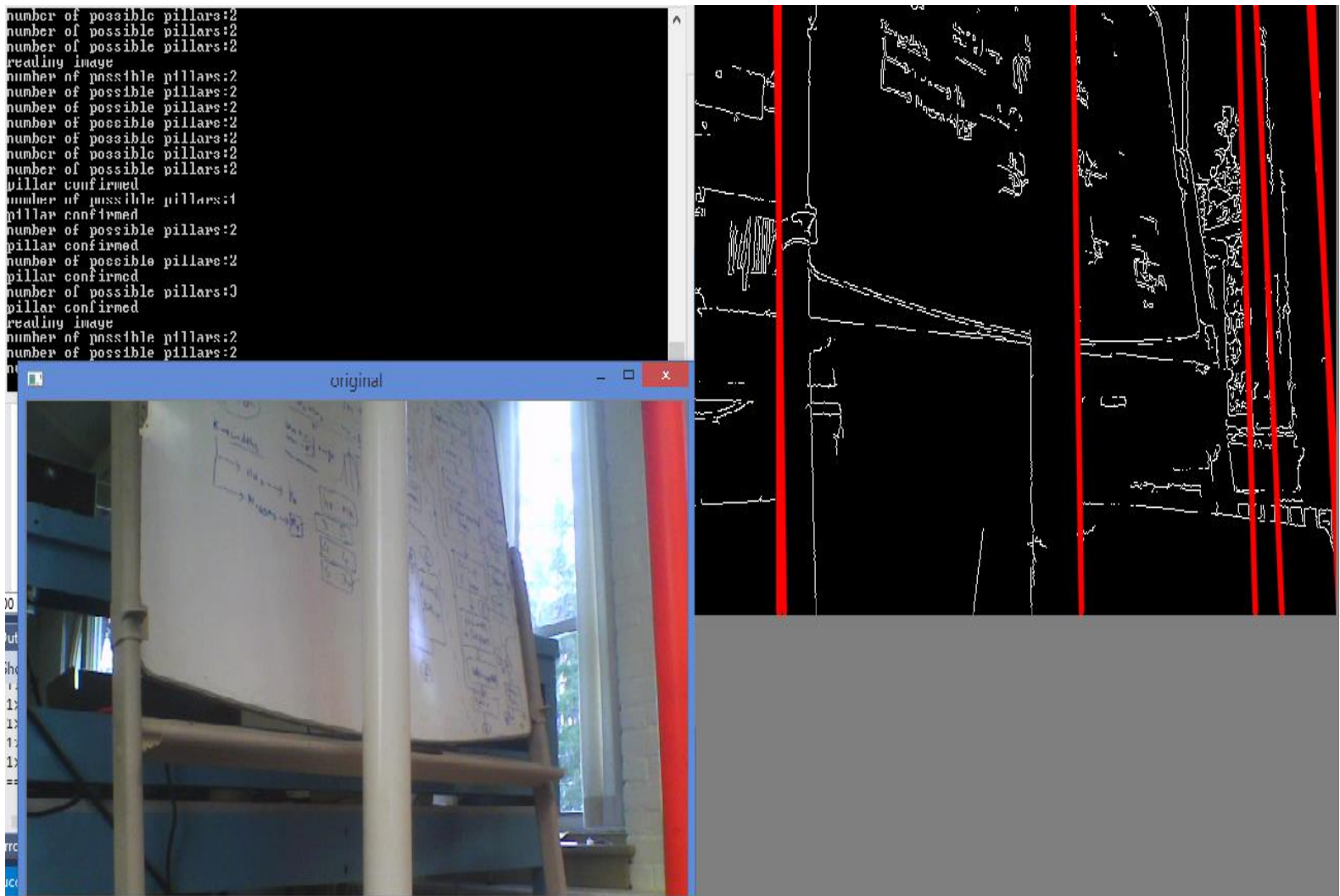


Figure 20: Pillar detection result1

In this case there are two pillars and some other features also that can be characterized by vertical edges. All vertical lines are detected but only those that are parallel are taken into consideration and based on that algorithm eliminates the false edges and determines the number of possible pillars. The number of detected possible pillars varies between 1 and 2. As we move the platform closer to the pillars, the parallel lines with the larger difference in rho values is consistently detected and hence, the message 'pillar confirmed' is given as the output. The white pillar which is the closer one is the detected pillar.

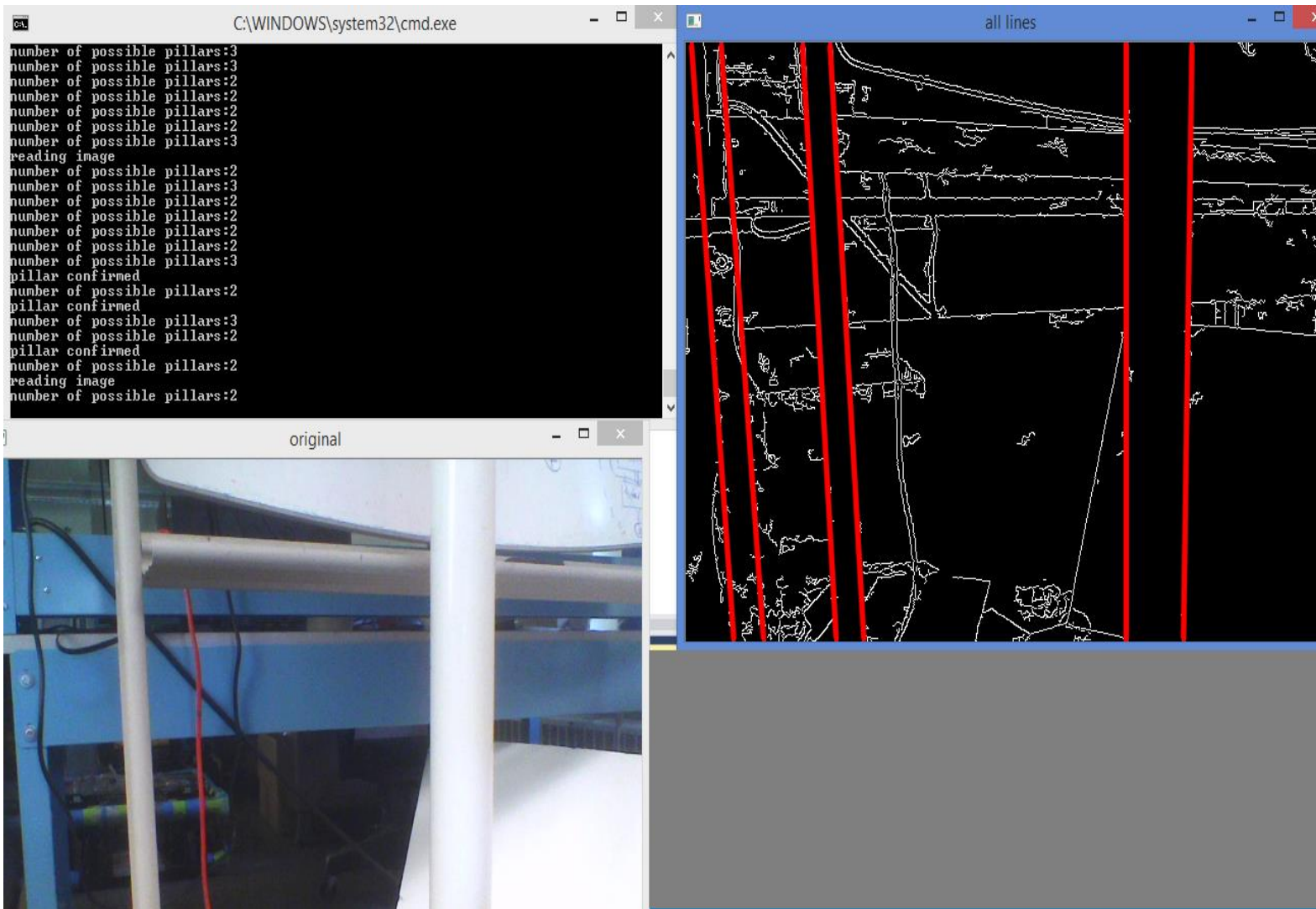


Figure 21: Pillar detection result 2

The number of detected possible pillars varies between 2 and 3. As we move the platform closer to the pillars, the parallel lines with the larger difference in rho values is consistently detected and hence, the message 'pillar confirmed' is given as the output. The white pillar which is the closer one is the detected pillar.

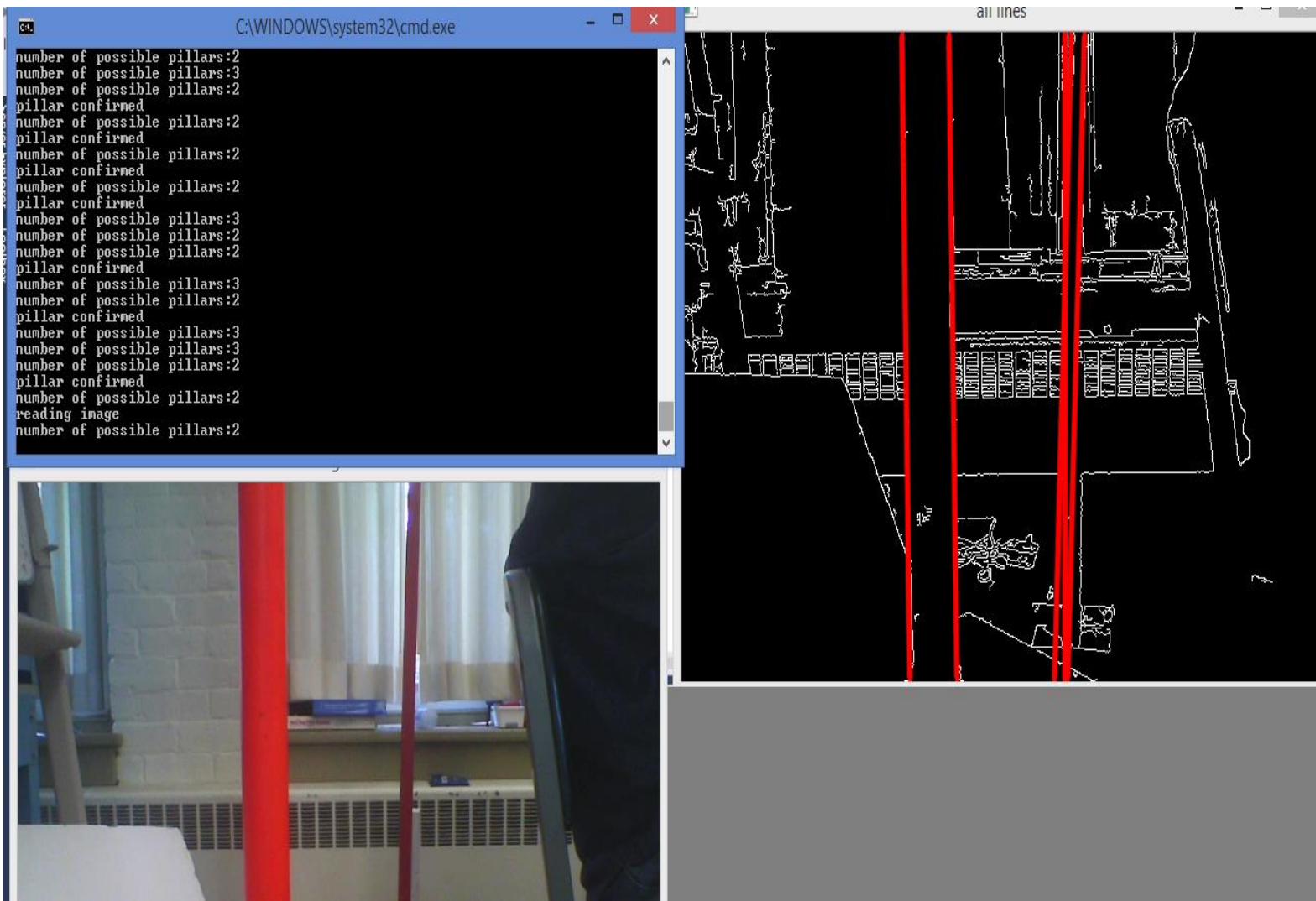


Figure 22: Pillar detection result 3

In this case there are two pillars. Consistently two possible pillars are detected and the larger one is confirmed. The number of detected possible pillars remains almost constant at 2. As we move the platform closer to the pillars, the parallel lines with the larger difference in rho values is consistently detected and hence, the message ‘pillar confirmed’ is given as the output. The orange pillar which is the closer one is the detected pillar.

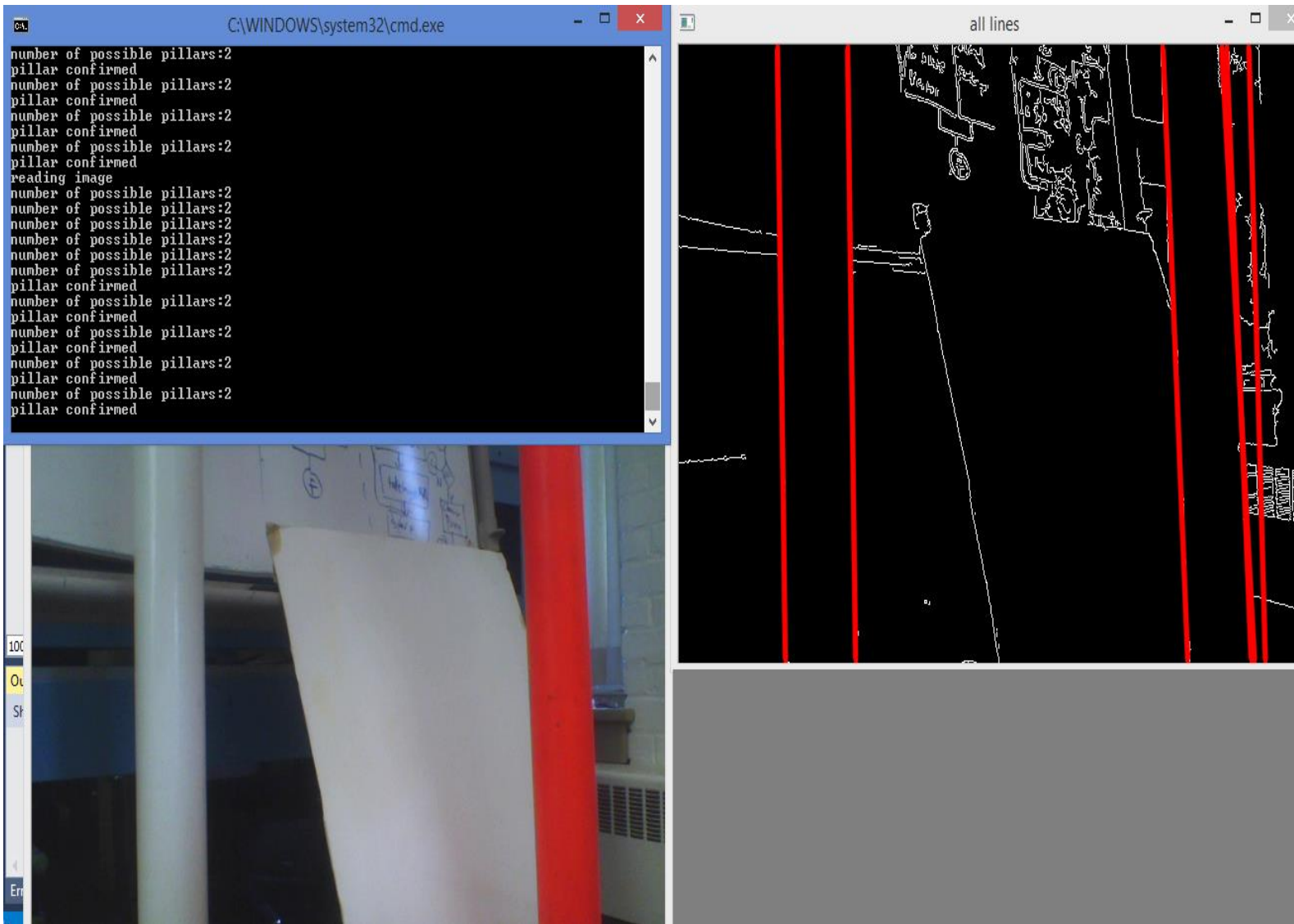


Figure 23: Pillar detection result 4

Again, in this case there are two pillars and consistently both are detected and tracked. Finally, the nearer one is confirmed. The number of detected possible pillars remains fully constant at 2. However, there is an issue here: Both pillars have same width in the real world. They are also at almost same distance from the camera. This is similar to the case described earlier by Figure 16 in Section 2.5. Pillar is detected, but we are not sure which one of them, and one of them is missed by our algorithm.

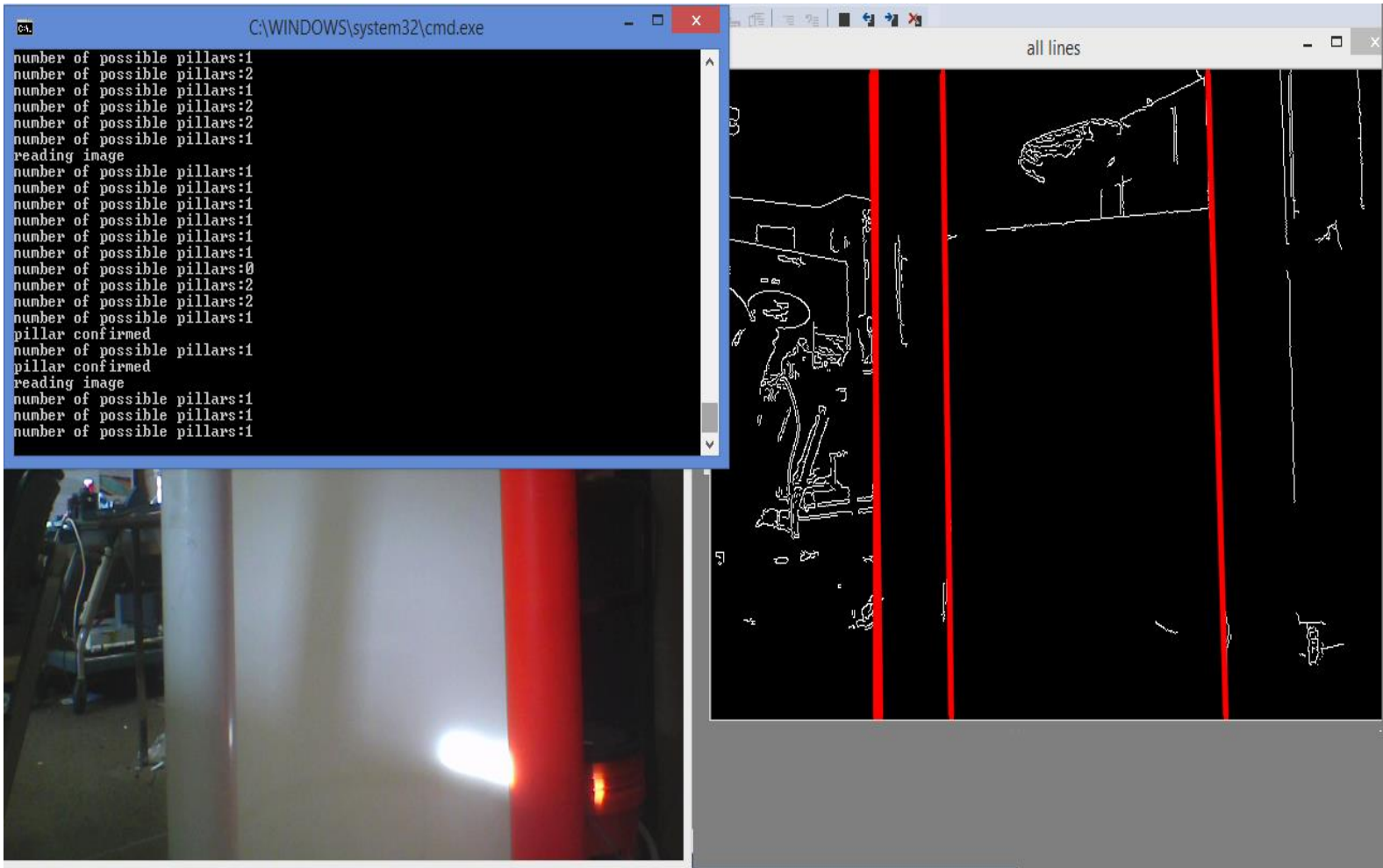


Figure 24: Pillar detection result 5

In this case there are two pillars. Only one pillar is detected and confirmed, due to inadequate lighting an edge of another pillar is not detected. The number of detected possible pillars remains almost constant at 1. As we move the platform closer to the pillars, only one is consistently detected and hence, the message ‘pillar confirmed’ is given as the output. The white pillar which is farther away from the camera is the detected pillar.

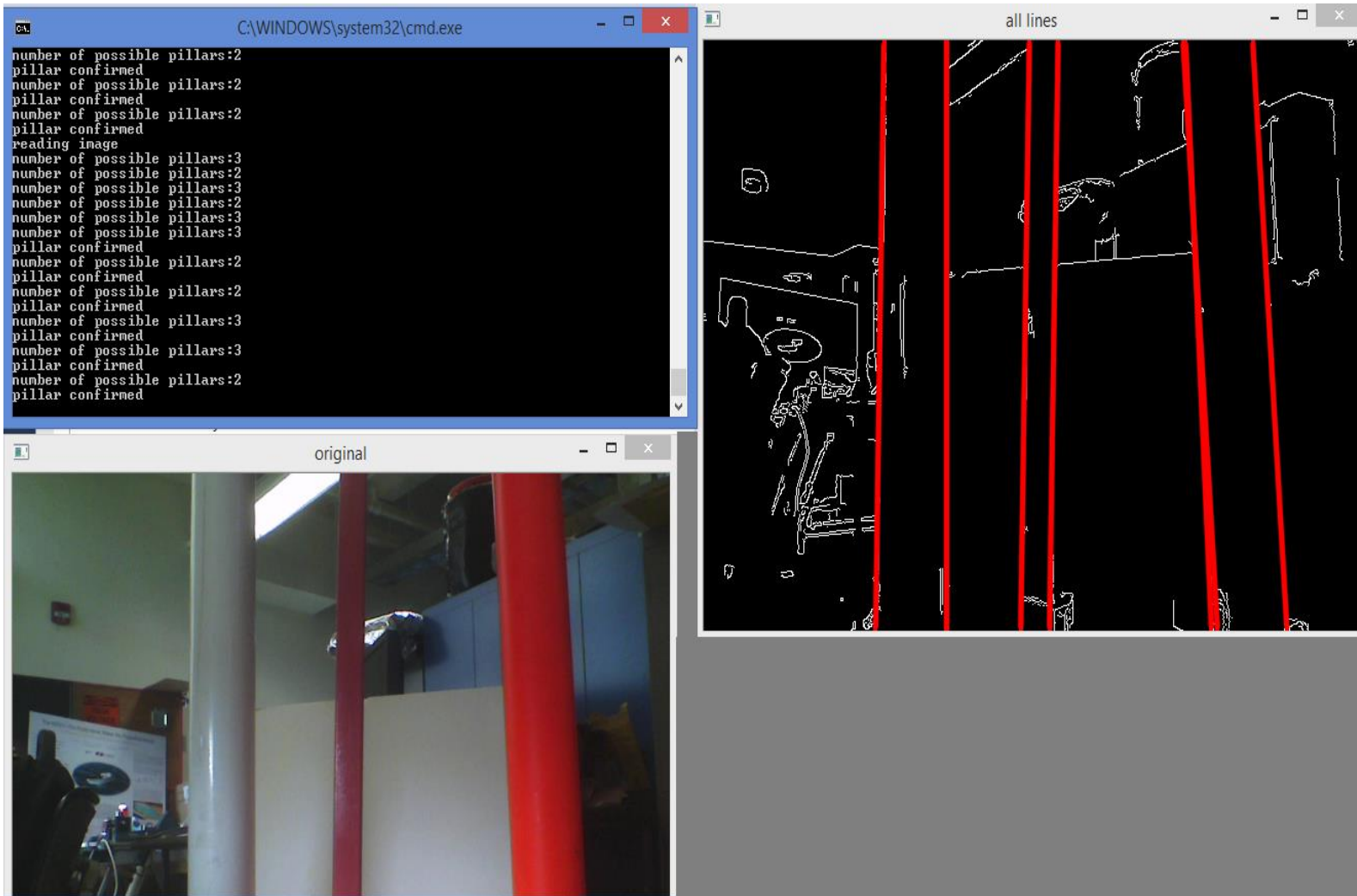


Figure 25: Pillar detection result 6

In this case there are three pillars. All vertical lines are detected and there are three pairs of parallel lines and all are taken into consideration by the algorithm. The number of detected possible pillars varies between 3 and 2. As we move the platform closer to the pillars, the parallel lines with the larger difference in rho values is consistently detected and hence, the message 'pillar confirmed' is given as the output. The orange pillar which is the closer one is the detected pillar.



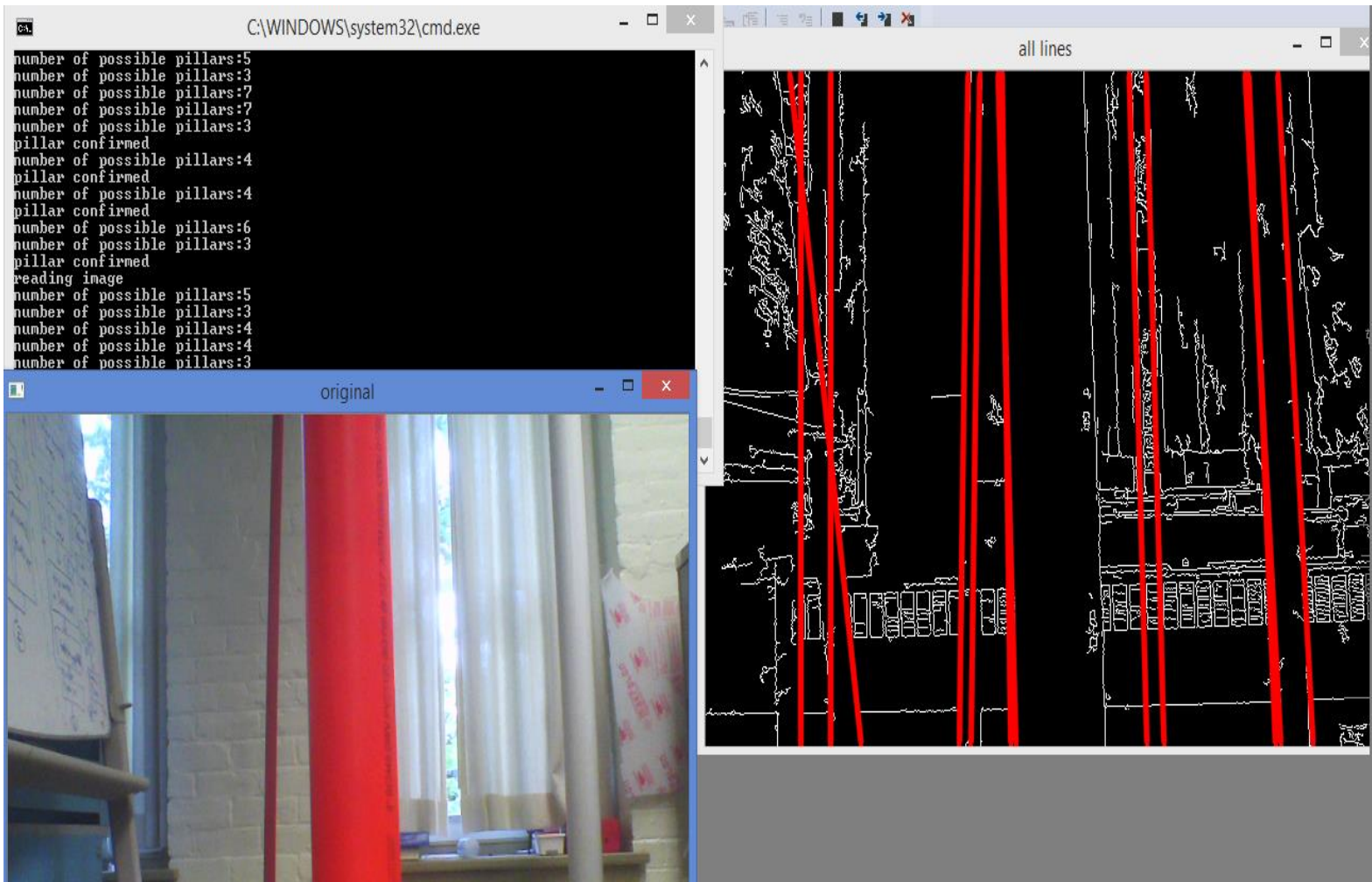


Figure 26: Pillar detection result 7

In this case there are three pillars and some other features also that can be characterized by vertical edges. All vertical lines are detected but only those that are parallel are taken into consideration. However, there is an issue here: There is a clutter of multiple edges. This is similar to the case described earlier by Figure 17 in Section 2.5. The number of detected possible pillars varies between 3 and 7. However, the Pillar is detected, message ‘pillar confirmed’ is given as the output. The threshold works correctly in this case and parallel lines wide apart are neglected and correctly, the orange pillar which is the closer one is detected.

## 2.6 Detection of Floor Edges

Since we are primarily building SLAM for indoor environments another prominent feature (besides a pillar) would be the floor edges. The hallway or a corridor are characterized by continuous parallel edges. However, unlike a pillar which are also characterized by two vertical parallel edges, the edges of the hallway do not appear parallel in the image. We have taken an image of one of the floor edges and performed Hough transform operation on it. We see that the edge detected is inclined at an angle which approximates to 45 degrees.



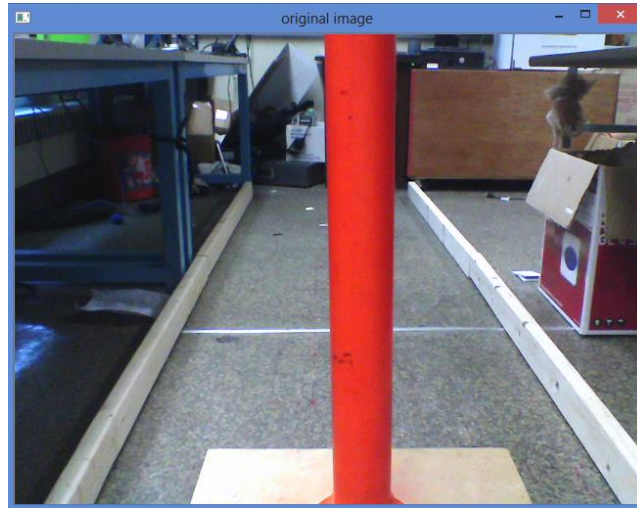
Figure 27: Inclined floor edge detection

Thus in order to use the Hough transform mechanism for detecting the floor edges, we again need to make the angle adjustments in the code. We had mentioned this earlier, that this angle adjustment would be the key to simultaneous detection of the pillar and the floor. The angle adjustment equation to be used in the code is as follows:

$$(((\theta < 0.785398) \ \&\& \ (\theta < 0.800)) \ || \ ((\theta > 2.420) \ \&\& \ (\theta < 2.4300)))$$

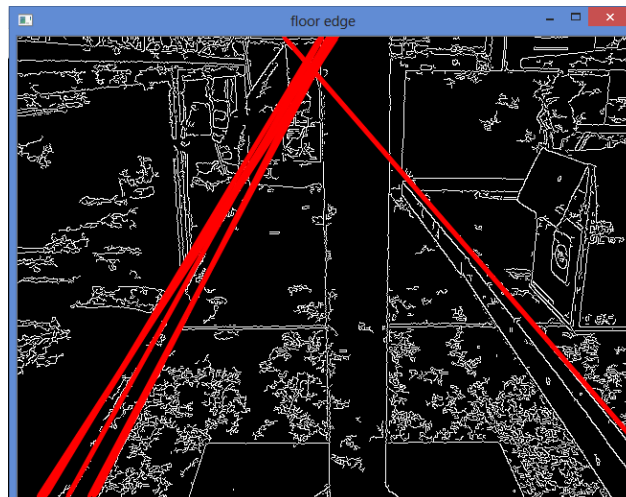
This equation is developed on the basis discussed in the earlier Section 2.3. It sets a range for the allowed line inclination for the floor edge line. This condition will allow the detection of hallways of different widths. In order to build a complete map of the hallway, detecting the floor edges would prove to be highly advantageous. The floor is the ultimate reference for the location of other features like the pillars, door edges, window sills etc. Knowing the location of these

features with respect to the floor edges would provide us with a robust map of the indoor environment. In the Figure 28, we see the original image containing pillar and a simulated hallway:



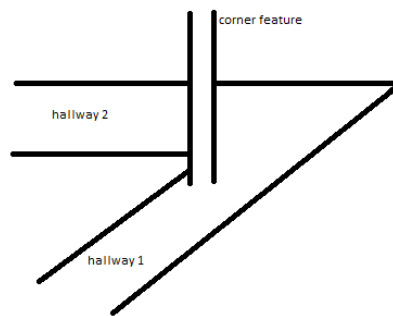
*Figure 28: Original scene that mimics a hallway*

The wooden bars in the image are arranged in a way similar to the actual edges of the hallway. The function returns the detected edges of these wooden bars. We get the image shown in Figure 29 as the output:



*Figure 29: Floor edge detection*

The change in line inclinations would mean a detected change in the hallway (assuming two hallways of different widths). If a feature location known with respect to one particular hallway is detected in another hallway, then that would constitute a corner location (intersection) of two different hallways. This is an important part required for determining map closure. When we move around an indoor environment and return to the original location (as detected by the presence of similar features that were detected earlier) we can achieve map closure, following which all features detected during the process are connected using their relative locations to each other. The corner features would help in connecting two hallways in the map. The following rough sketch shown in Figure 30 illustrates this situation.



*Figure 30: Corner feature*

Now, in order to calculate the relative location of the feature with respect to the hallway, we begin with tagging the first set of inclined lines as hallway 1. All the features detected with these constant set of lines are assumed to belong to this hallway. The floor edges are parameterized using their depth from the camera as described in Section 1.5. As we calculate the distances of these features from the camera, as well as their lateral shifts, it signifies the feature location within that hallway.

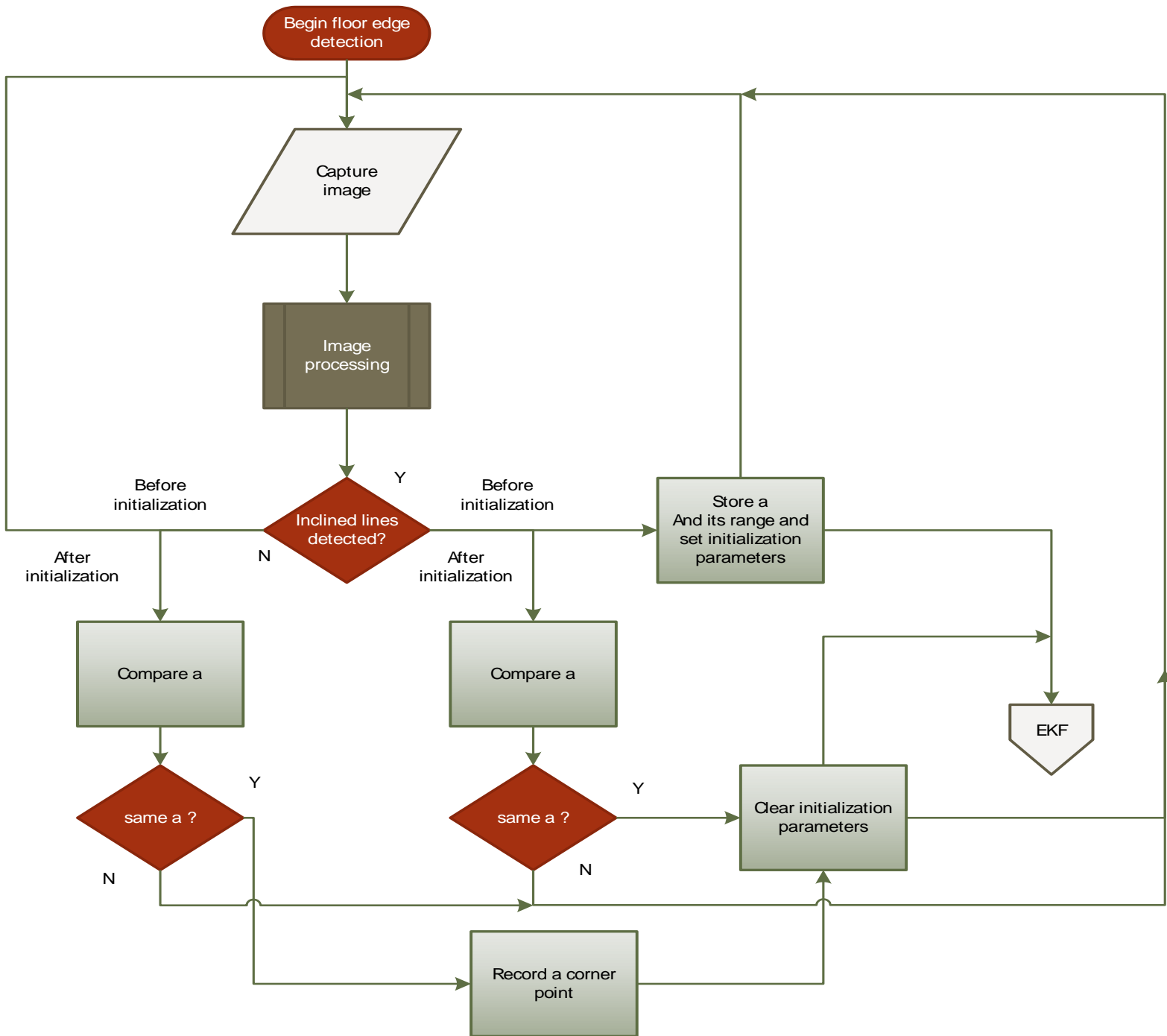


Figure 31: Floor edge detection algorithm

Now, with this background we can discuss the entire algorithm for floor edge detection. As shown in Figure 31, the algorithm is similar to the pillar detection algorithm in some aspects. However, the line orientation range has been changed to detect inclined lines and we don't have

a learning-checking phase here. Instead, we continue the detection iteration until a break in the inclined lines is detected. Whenever the break is detected the robot pose  $\alpha$  is checked. If this angle is almost similar (within the range) of the previous angle when the lines were first detected, then it implies change in the hallway direction. With the first detection of inclined lines, the feature (the floor edge) is initialized (as described in Section 1.5) and the break would determine change in hallway and hence would call for a new feature initialization. The feature initialization (following feature detection) for both pillar and the floor edge was discussed in Section 1.5.

## **2.7 Synchronization with delayed initialization and EKF update**

Our primary research goal was to develop feature detection algorithms in software for monocular SLAM and then to create hardware implementations of portions of those algorithms in order to improve overall system performance. In developing such algorithms however, it is essential that they work in synchronization with the overall EKF-SLAM system, to build an environment map. That means, recognized features have to be initialized and added to the map (state vector and its co-variance matrix) and they must be updated subsequently in accordance with the predict-correct model of the EKF. We are proposing monocular SLAM based on the concept of delayed initialization. We know that in delayed initialization, candidate features are initialized based on certain constraints such as sufficient baseline generation and creation of adequate parallax. This delayed initialization model works in synchronization with our feature detection algorithm as shown in the flow chart presented below in Figure 32.

When the detection cycle begins, a feature is taken to be a candidate for initialization. We have not confirmed it to be a pillar that we are consistently tracking, but we assume it to be one. The parameter extraction for the feature begins based on the combination of the observation

model explained in Section 1.5 and the delayed feature initialization explained in Section 1.4. When the detection cycle ends and we have confirmed it to be a consistently tracked pillar, then the feature is accepted for further processing, else it is purged. Once, it is accepted for further processing, based on the conditions explained in [13], it is initialized and added to the map. Once the feature is added to the map, the EKF is updated.

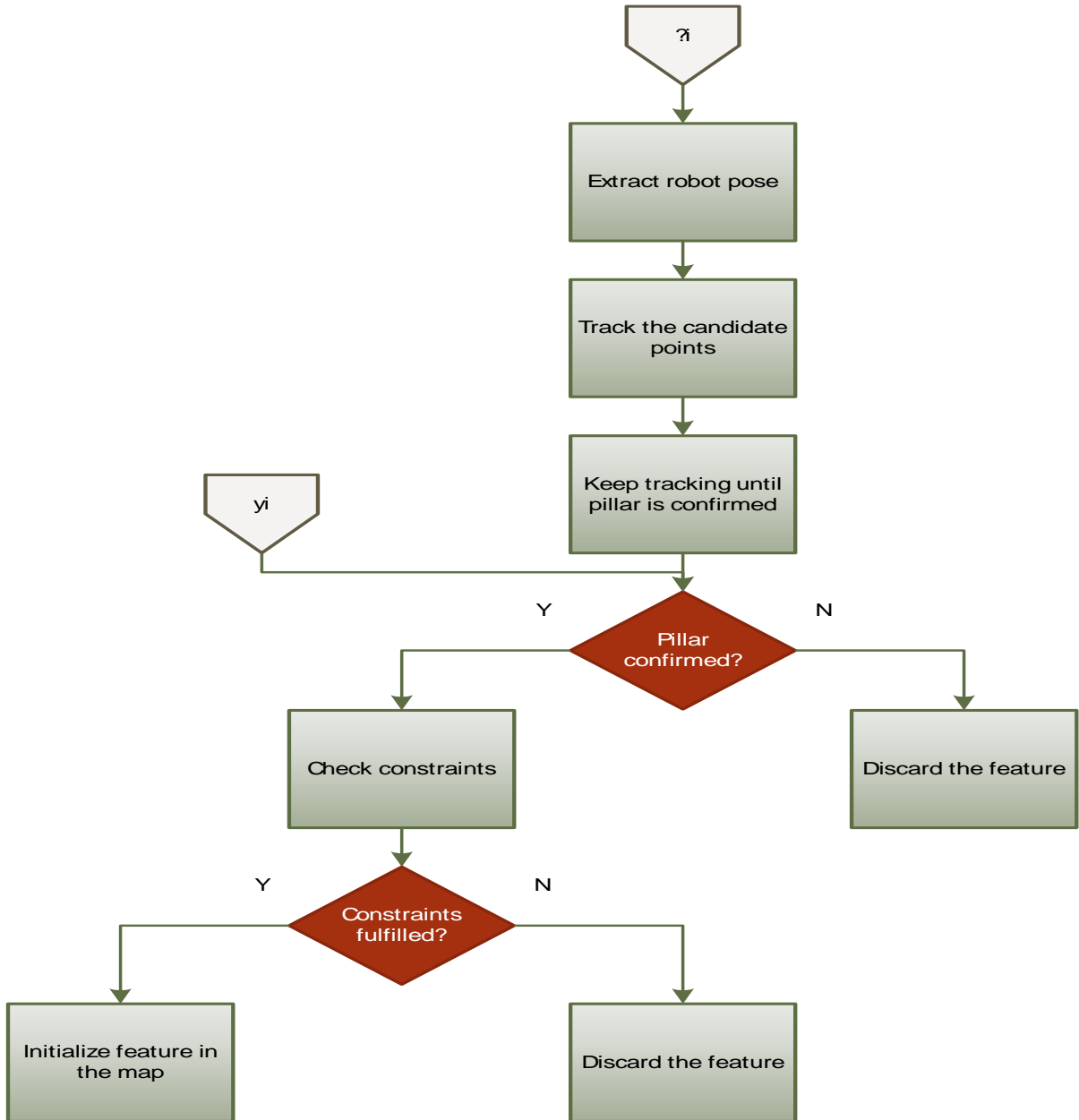


Figure 32: Delayed Initialization

Similarly in case of floor edge detection, the initialization is done when the detection cycle begins. In the case of floor detection there is no purging, the inclined lines are unconditionally initialized, which is based on assumption that the only possible pair of lines inclined simultaneously around  $45^\circ$  and  $135^\circ$  has to be the floor edges (or outer bounds for any indoor environment).

The initialization takes place according to the delayed initialization process explained in [13] & [15]. The next initialization takes place only when a break in the lines is detected at which time a new detection cycle begins. The whole process is summarized in Figure 33.

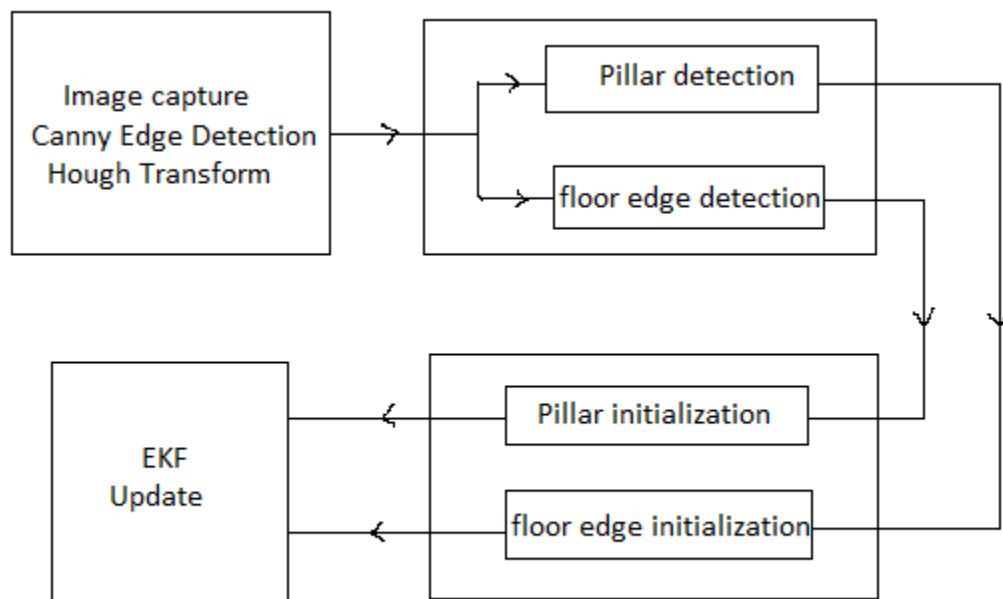


Figure 33: Synchronization with EKF



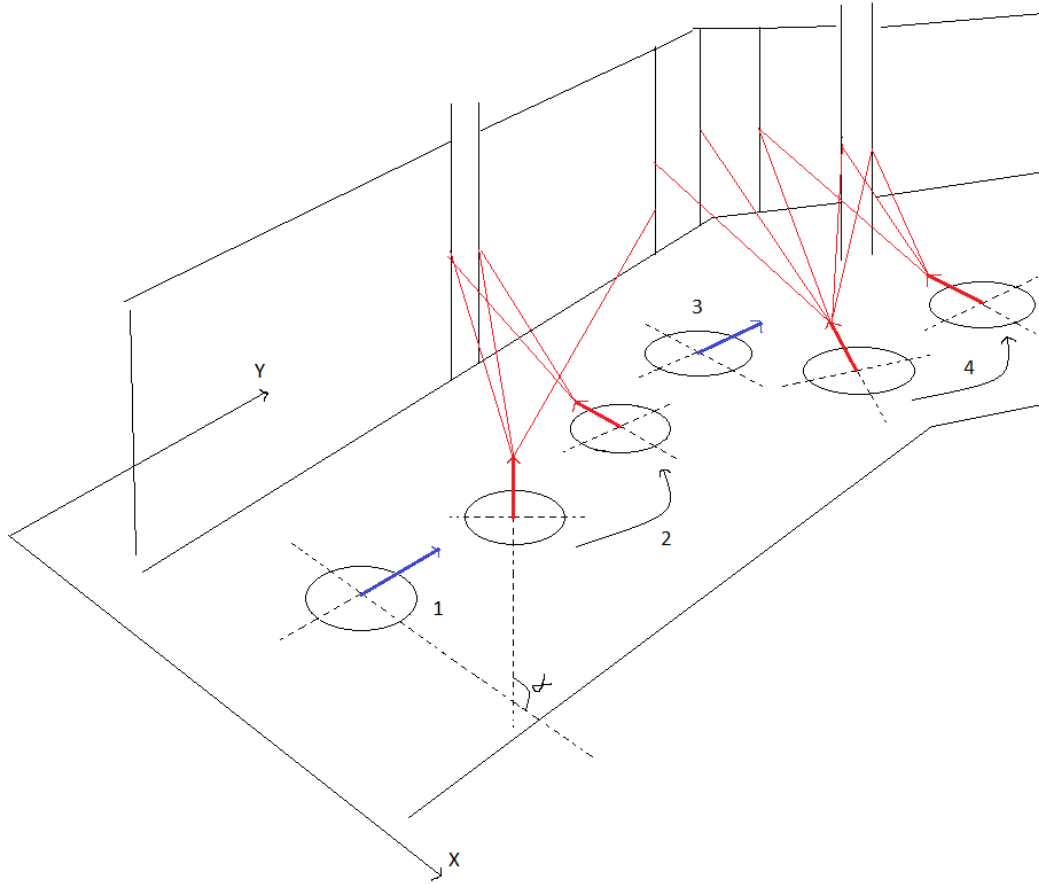


Figure 34: Example case for synchronized feature detection and EKF update

In Figure 34 we have considered an example case for a bearing-only SLAM implementation. A hallway-like environment is considered and we assume the robotic platform to move through it. We have shown the X-Y plane and the orientation of the platform with respect to the world coordinate system. Each position of the robot (robot pose) is described as  $(x_k, y_k, \alpha_k)$ . Events occurring at each point are described as:

1] The inclined lines are observed, which represent the floor edges. These floor edges are initialized at this point. If in subsequent observations, there is no break in the inclination detected, the floor line is encoded as  $y_i = (x_i, y_i, \alpha_i, d_i)$ .

2a] Number of vertical lines are detected. Amongst these lines all combinations of parallel lines are possible pillars. All of them are taken as a candidate for initialization and the

procedure for delayed initialization is initiated, but pillar is not yet confirmed. The actual pillar which is to be tracked is the nearer one (which is not yet known), however, Hough transform will also detect the wall edge as shown (and this cause for detection multiple parallel lines). Applying threshold will probably remove it. If not, the continuous frame capture from this position to the last one in the detection cycle will eliminate it and the pillar will be confirmed by our algorithm.

2b] when the possibility of a pillar was identified in step 2a, we had immediately begun the procedure for delayed initialization. Once the pillar is confirmed, the baseline and parallax angle are checked (i.e.  $|b| > |b_{\min}|$  and  $a > a_{\min}$  [refer 13]), and if the condition is satisfied the feature is initialized into the map encoded with its inverse depth  $y_i = (x_i, y_i, \alpha_i, \rho_i)$ .

3] In this position a break in the inclined lines is detected with the same  $\alpha$  and hence, a new detection cycle has to begin for floor edges. However, this will not take place immediately; over subsequent movement towards the new hallway, the inclined lines will be detected within the specific angles and then the new feature will be initialized.

4] This is a very noisy case where threshold will likely fail. The pillar may be confirmed if it fits within the range built in the learning phase, else it will be discarded altogether.

After each of the above cases, the EKF is updated and a new co-variance matrix is calculated. This results in an incremental map building as robotic platform proceeds in the environment.

## 2.8 Initial experiment in map building with one known feature

The feature detection experiment in this section is based on one known pillar (of known width) and then mapping the rest of the landmarks (with unknown widths) with respect to this known pillar. Also, all our parameters lie on single plane, which is the plane of the floor. We assume a fixed camera pointing in a constant direction as our sole sensor. Our three parameters

are: distance from the pillar  $d$ , Horizontal shift from the pillar  $s$  and the angle  $\theta$ . This is where our first landmark in the map, i.e. a pillar with known width comes into picture. We have pre-calibrated the distance and the lateral shift of the camera with respect to the pillar in terms of changes in values of  $\rho$ , which we call ‘delta rho’ as in the earlier sections.

This delta rho is a key parameter which will ultimately help us determine the location of the feature with respect to the camera and that in turn will help us create the state vector for the Extended Kalman filter. As the pillar moves away from the camera the width of the pillar in the image will be decreasing. Similarly, as the pillar moves closer to the camera its width in the image will be increasing. These trends will be reflected in the values of delta rho. Theoretically, the value of delta rho should increase proportionally or decrease proportionally with an increase or decrease in the image width of the pillar.

We now experimentally observe the values of the delta rho corresponding to the change in distance of the pillar from the camera. We perform a calibration procedure for a known pillar. In this case we calibrate from 30 inches (76.2 cm) till the distance at which the code stops detecting the pillar. We find that until the distance reaches 100 inches the system is able to detect the pillar. Beyond that range, the Hough transform approach becomes unreliable. The consistency of detection is high only till 60 inches (152.4 cm) beyond which the detection becomes erratic and in the range of 90 to 100 inches (228.6 cm to 254 cm) the detection is not reliably consistent and will cause reduction in the accuracy of distance value calculated by the function. The table of delta rho values calculated by the function corresponding to the distance is provided in Figure 35:

distance in cm	distance in inches	delta rho	pillar detection accuracy
76.2	30	69	high accuracy
88.9	35	60	
101.6	40	50	
127	50	40	
152.4	60	33	
177.8	70	28	moderate accuracy
203.2	80	24	low accuracy
228.6	90	22	
254	100	19	

Figure 35: distance-delta rho observation<sup>3</sup>

From, the above observations we realize that the change in delta rho is not a linear function of the change in distance. The curve plotted with distance on the X-axis and the delta rho on the Y-axis reveals a curve that approximates an exponentially decaying function. In this case we make use of the curve fitting tool of MATLAB. The two vectors 'x' and 'y' contain one dimensional values that we have obtained in the table above. In the 'cftool' GUI we choose the exponential function for our fitting and we get the following results shown in Figure 36.

MATLAB was used to calculate the equation constants that will help us create the calibration equation giving results that are very close to the original values. However, there will be a margin of error for a certain range of values. As seen in the Figure 36, the range of 30 to 70 inches (76.2 to 177.8 cm) has the best possible match between the actual values and the fitted values. The calibration equation thus derived is as follows:

$$d = \left[ \frac{-\log_{10} \left( \frac{\Delta \rho}{151.8} \right)}{0.0108} \right]$$

<sup>3</sup> 1 inch = 25.44 mm & 1 cm = 0.39370 inches

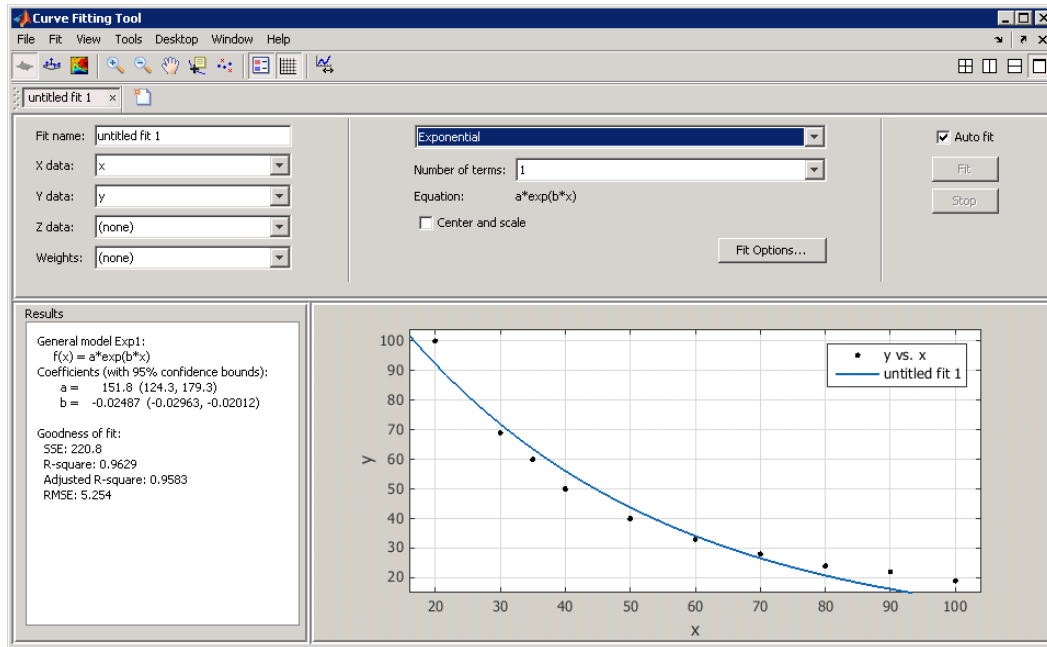


Figure 36: distance calibration

In order to create a two dimensional map, we require at least two parameters: one is the change in the perpendicular distance of the feature (in our case a pillar) with respect to the camera and the other is the lateral shift of the pillar with respect to the camera. The equation above gives the values of the first parameter. With help of that equation we can easily calculate the change in distance over a period of time (or over a number of frames). As we will see in the later sections, however, we also have to determine the lateral shift of the pillar. Knowing both, would allow determining both the x and y co-ordinate of the feature with respect to camera and enable a basic map building.

To determine the location of the pillar laterally, we note that as the pillar shifts leftwards, the position of the pillar in the image moves towards the origin and this causes a decrease in leftmost<sup>4</sup> rho value. Similarly, if the pillar moves rightwards the value of leftmost rho increases. This fact will help us calibrate the lateral shift of the pillar in terms of rho value.

<sup>4</sup> OpenCV 'HoughLines' function assumes the origin to be at the top-left corner of the image.

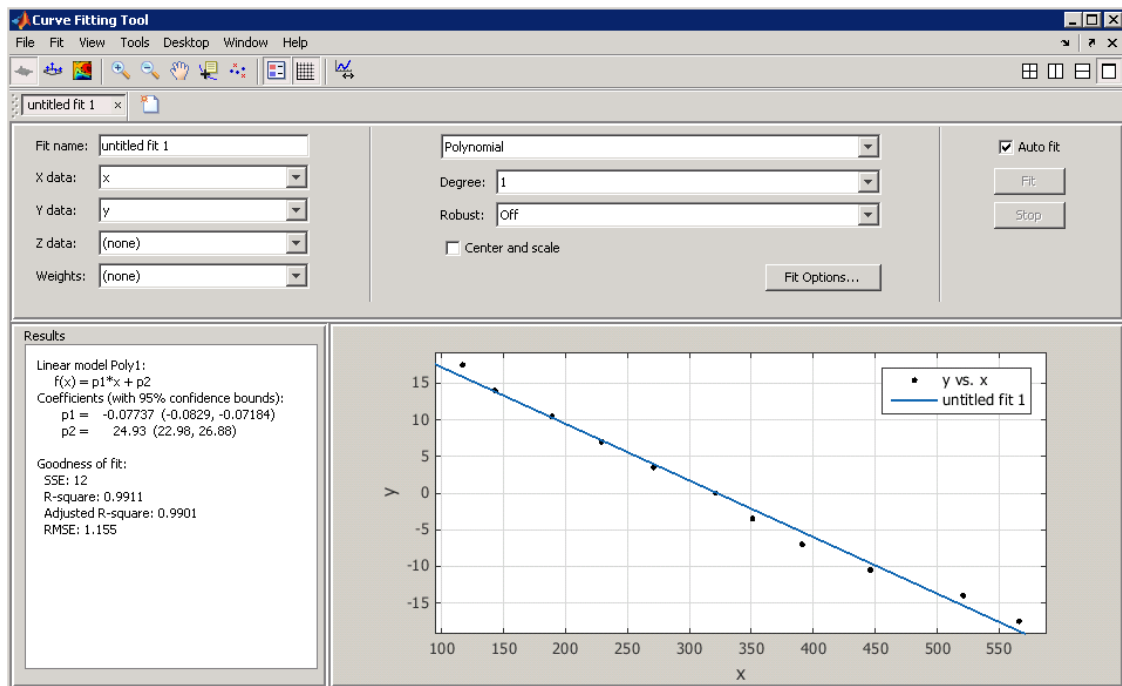
As usual, Hough transform is used as a tool to detect vertical parallel lines which are assumed to be a pillar in the beginning. The ‘rho’ values of these parallel vertical lines are used to detect all possible widths (differences in rho values of all detected vertical lines), all of which could be potential pillars. It is important to note that ‘rho’ and ‘theta’ are just parameters in the Hough transform voting space and hence, the widths calculated through the difference in ‘rho’ values are all virtual widths. They have to be mapped to some reference to be considered as useful parameters. We use the Hough transform mechanism for this purpose as well. As described earlier Hough transform helps us detect edges of the pillar, however, the image containing the detected edges is noisy. Each of these edges (along with the noisy false edges) of the pillar has a value of rho associated with it. We use the same algorithm as used earlier and eliminate non-parallel edges leaving us with only pairs of parallel edges and an array containing the corresponding rho values of these remaining edges. We take the smallest of all the remaining rho values as our calibration parameter. In the rough sketch shown earlier (refer to Figure 12), we see that the Hough transform mechanism returns four rho values. Our algorithm will eliminate rho1 and rho2 as they are non-parallel leaving the two values rho 4 and rho 3. Out of these, rho 4 has the smallest value and thus becomes our calibration parameter.

We perform the calibration experiment that we have performed earlier. In this case however, we mark the lateral distances such that the perpendicular distance from the camera is marked as 0 which will be our reference center. As we move rightwards we increment the lateral shift in steps positive of 3.5 inches (8.89 cm) and the leftwards shift would be in terms of negative 3.5 inches (8.89 cm). The value of rho 4 (that is the leftmost valid rho) corresponding to the lateral shift from the reference center is calculated as in table 37:

distance in cm	distance in inches	leftmost rho
44.45	-17.5	566
35.56	-14	521
26.67	-10.5	446
17.78	-7	391
8.89	-3.5	351
0	0	321
8.89	3.5	271
17.78	7	229
26.67	10.5	189
35.56	14	143
44.45	17.5	117

Figure 37: lateral shift-leftmost rho observation<sup>5</sup>

Now, we use the curve fitting tool of the MATLAB to generate the equation for this set of observations. The resultant curve fitted over our observations of lateral position is fairly linear as shown in Figure 38:



<sup>5</sup> 1 inch = 25.44 mm & 1 cm = 0.39370 inches

*Figure 38: lateral shift calibration*

The calibration equation for the lateral shift would be as follows:

$$d = -0.07737\Delta\rho + 24.93$$

Now, we have the two necessary parameters to create a two dimensional map: the perpendicular distance from the camera and the lateral shift with respect to the reference center.

The next step towards a building comprehensive map would be being able to display a basic two dimensional map of the feature (in our case pillar) and the path traced (movement of camera with respect to the feature). At this stage we do not store the map since it requires a map update by the Extended Kalman filter which we have not designed. However, we must be able to trace a path with respect to the pillar that we have been able to detect earlier and display it in form of two dimensional map.

Now, we have the two essential parameters for a two dimensional map: the perpendicular distance of the pillar from the camera and the lateral shift of the pillar with respect to the reference center. The update in the map is done in a period over 100 frames. That is we track the changes on frame by frame basis based upon the changes taking place over a period of 100 frames and then we update the map. These 100 frame periods repeat continuously. In each case the linear path travelled from the location at the end of earlier iteration to the end of current iteration is updated and displayed on the map window along with the feature location with respect to the travelled path. The 100 frame period was chosen by trial and error to accommodate the considerable changes that can be detected and updated. Sporadic changes taking place during this period are smoothed out. This obviously makes the map less sensitive to rapid and random changes but allows it to respond well to slow changes. This assumption means that we are



presuming that any changes taking place are smooth, and slow enough to be accommodated in the 100 frame duration.

The change in perpendicular distance is calculated based on the algorithm detailed in Section 1.3. As described in Section 1.3, this algorithm also eliminates false distance readings due to the noise incurred by the Hough transform. For the first frame, the distance calculated is stored in a variable. From the second frame onwards the new distance is calculated from each frame and is compared with the value of the stored variable. If the distance is within the error range of  $\pm 1$  inches (2.54 cm) then we store its value in a second variable. Each time this error condition is true we keep on adding the new distance to this second variable. Finally, we calculate the average of all such occurrences. A count is maintained every time the new distance is added to the second variable which helps in the calculation of the average. Now, if the change in distance is negligible, it will be interpreted as no change in distance, else the value of distance in the final frame is compared with that stored in the first frame to give us the change in distance. A negative value of distance change indicate that the new distance is less than the old distance in which case it is a backward move (towards the camera) else if it is a positive value then it is interpreted as the forward shift (away from the camera).

Now, the second parameter, lateral shift is calculated simply by subtracting the position at the first frame and that at the last frame. It is not necessary to average out this parameter because it has been observed that lateral shift calculation does not show considerable error. This is because the calculation of lateral shift is based on detection of single, leftmost, edge as compared to double edges as in case of perpendicular distance. Now, if the lateral location value is less than the new location it means the pillar has shifted to the left else it has shifted to the right.

Based on the above data we can display the map. The map display has to begin with the initial positions. In our case the pillar is the feature whose initial position with respect to the camera is to be determined before we can start tracing the path. This is done in the following way: The distance  $d$  of the pillar from the camera calculated in the first frame is stored. Then after iterating through the first 100 frames, we begin marking the initial position of the pillar and the camera. The initial position of the detected feature (that is the pillar) is chosen arbitrarily on the map as the location  $(100,100)$ . The position of the camera is then calculated as  $(100,100+d)$ . This location of the camera is the starting point of the path. After the next 100 frame iteration, the new camera location is calculated on the basis of the distance travelled and the lateral shift incurred. The path is created connecting  $(100,100+d)$  and these new coordinates are recorded. There are four possible cases as illustrated in Figure 39 below.

Let  $s$  be the shift and  $d$  be the initial distance traveled and  $d'$  be the distance travelled in during the next 100 frame iteration. Based on this the following rough sketch will explain the four possible situations:

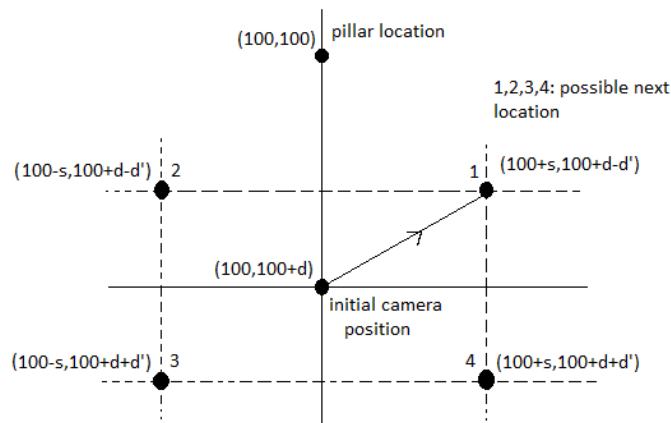
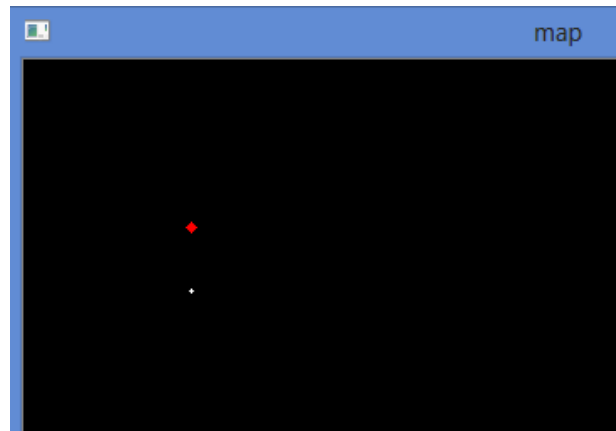


Figure 39: sketch for map display

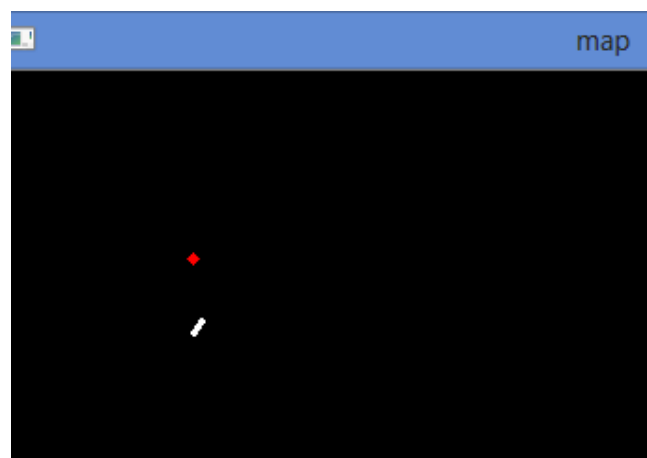
These calculated coordinates become the starting point for the following iterations and the new coordinates calculated over that iteration become the end points for the next iteration.

The line connecting these two pairs of coordinates approximates the path travelled in the prior iteration. The map is developed continuously, over several iterations. The circle and line functions of OpenCV are used in the code to develop the map display. The initial location in the output of the map is as shown in Figure 40:



*Figure 40: Initial feature position*

In the map window illustrated in Figure 40, the red dot signifies the location of the pillar and the white dot is the initial location of the camera. Now, when we move the camera, the path traced with respect to the pillar is reflected in the following map window:



*Figure 41: display of first shift in camera with respect to known pillar*

Now, once this first feature is mapped all other features can be mapped with this known pillar as a reference.

# Chapter 3

## Feature Detection Using FPGAs

### 3.1 Hardware Acceleration Using FPGAs

The field programmable gate arrays (FPGA) are widely used for the purpose of testing the hardware designs. However these days they have gained popularity for commercial applications and some products do exist in markets which combine a processing core with an FPGA fabric. A recent example of this is Samsung Galaxy S4 smartphone which uses the Lattice LP1K36 1K mobile FPGA.

Understanding the properties and basic functioning of FPGAs is extremely crucial to this thesis. These important properties make it an ideal choice for optimization of timing performance of our SLAM application.

FPGAs are generally referred to as ‘soft hardware’ [26]. FPGAs are built from cells each of which can be programmed to realize a logic function. They consist of generally three components: reconfigurable circuit blocks which can be programmed for a particular logic function, programmable interconnects, and I/O pins. Each logic block is a combination of a combinational function and registers. The combinational function is implemented using a look-up table which is implemented in SRAM. Using these logic blocks FPGAs can be used to create customized logic for each algorithm or application. Each functional unit is optimized to perform specific tasks. Also more than one unit can be executed concurrently. FPGAs also have an added advantage of being reconfigurable. They can achieve parallelism to meet their design objectives

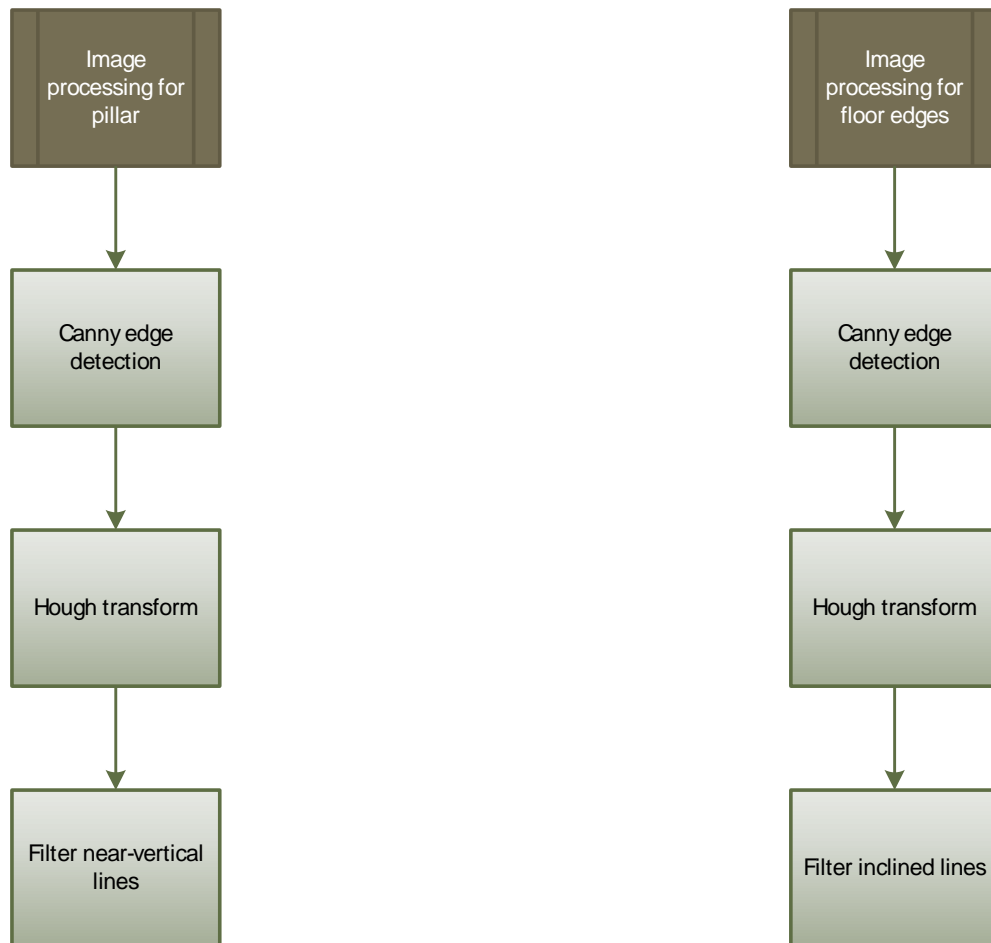
and their functionality can be changed to allow implementing algorithm changes. Further, since algorithms can be implemented directly in hardware, FPGAs can be faster than equivalent software executed on a microprocessor. This is in spite of the fact that FPGAs often have lower clock frequency than microprocessors. This potential performance improvement occurs for three reasons: FPGAs have no instruction processing overhead and allow bit level control of operation, they can transfer data at high speed on-chip with negligible latency, and they allow parallel hardware elements to exploit parallelism in an algorithm. These facts are expected to make FPGA beneficial for our application. Being fast as compared to a microprocessor, and at the same time having lower clock frequency, is exactly what one would desire if they were to design a system with better time and power performance compared to a microprocessor-based system

The two major players in FPGA field are Altera and Xilinx [26]. They both have various families of FPGAs and development boards that are available in the market, along with various coding and simulation software. In our research we use the Xilinx Spartan 6 and Xilinx Spartan 3 as our target FPGAs.

An important architectural feature of embedded computer system is the usage of hardware accelerators. Most of the functions in a computing systems are well known. Hence, instead of using a general purpose processor, the system can be split between a processor performing certain number of tasks, while exporting certain features to specialized hardware. For example, many applications use a separate DSP for signal processing tasks along with a traditional RISC based processor to do other general tasks. This is where FPGAs play an important role. The features of FPGAs mentioned above, allow building special-purpose a hardware accelerators on the FPGA fabric. Such hardware accelerators can increase the efficiency and performance of an embedded system.

### 3.2 Image processing for feature detection on FPGAs

We have developed the feature detection algorithm and implemented it using C++ and the OpenCV Library. Now, we move on to the second part of our research which is developing Hardware accelerators for feature detection using FPGAs.



*Figure 42: Image processing for Canny edge detection and Hough Transform*

Figure 42 contains a flowchart that illustrates the image processing sub-process within the feature detection algorithms [refer to Figures 19 and 31]. The Canny edge detection and Hough transform take place iteratively for almost infinite duration (as long as the system is

running). Also, these operations are done for both pillar and floor edge detection. Therefore, it makes sense to offload these algorithms from software and implement them using an FPGA-based hardware accelerator. The possibility of designing a pipelined architecture on FPGA makes it an attractive proposal, since our ultimate aim is to design a time-efficient algorithm. This is made possible by exploiting parallelism within the FPGAs. In the next chapter we go into the details of the Hough transform implementation on FPGAs, followed by the implementation of Canny edge detection. We present the results of comparing the performance of the hardware-accelerated and software-only system implementations in the final sections.

### **3.3 Canny edge detection using FPGA**

Although it isn't necessary to perform edge detection before performing Hough transform on the input image, it is highly advantageous to do so. This is because, the Hough transform, is used to detect shapes such as lines and circles within an image and these shapes generally describe edges of the objects within an image. Thus, we can eliminate pixels that constitute the surface of the objects and focus completely on the edges, and this makes performing Hough transform on the image easier. Therefore, Canny edge detection, by convention, is performed prior to the Hough transform operation.

#### Canny Edge Detection Mechanism

Canny edge detection generates a binary image, which has intensity of  $1$  where it detects an edge and the intensity of  $0$  elsewhere. Our Hough transform module has one bit input port, as we have seen earlier, which represents the intensity of the corresponding input pixel coordinates. As we have seen in Section 2.2, there is a function for Canny edge detection in the OpenCV library and that allows performing edge detection without going into the details of its mechanism.

However, in order to implement the algorithm in hardware, we have to understand its details. In the following sections we will begin with the concept of edge detection in image processing, and then move towards the actual Canny edge detection algorithm, and finally explain the assumptions and the approximations within the algorithm to make it hardware implementable.

Edge detection refers to the process of identifying and locating sharp discontinuities in an image, which actually represent abrupt changes in pixel intensity. The change in intensity characterizes boundaries of objects in an image. Ideally, a change in intensity is assumed to be a very sharp change and it is called a step edge. However, step edges are rare in real images. Because of low-frequency components or the smoothing introduced by most sensing devices, sharp discontinuities rarely exist in real signals [1]. For the sake of modeling an edge detector, we consider an ideal step edge corrupted by a Gaussian noise process. In practice this is not an exact model but it represents an approximation to the effects of sensor noise, sampling and quantization. Let us consider an ideal step edge, and its convolution with a Gaussian function. The result is a smoothed edge which is represented as shown in Figure 43.

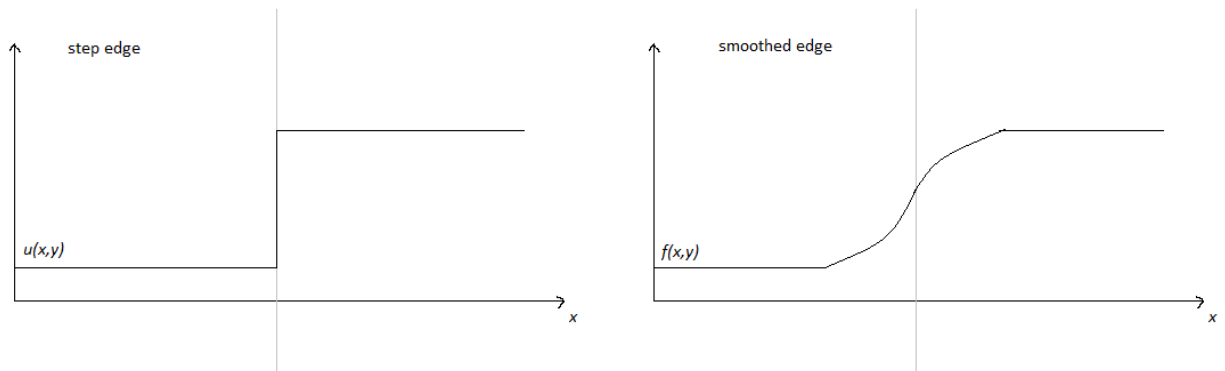


Figure 43: Smoothed edge [1]

We can see, there is now a gradual change in intensity. Now, we can detect edges in two ways: First is, we calculate the gradient of this smoothed curve, which is its first order derivative. The edge is then the maxima of the curve as shown in Figure 44:



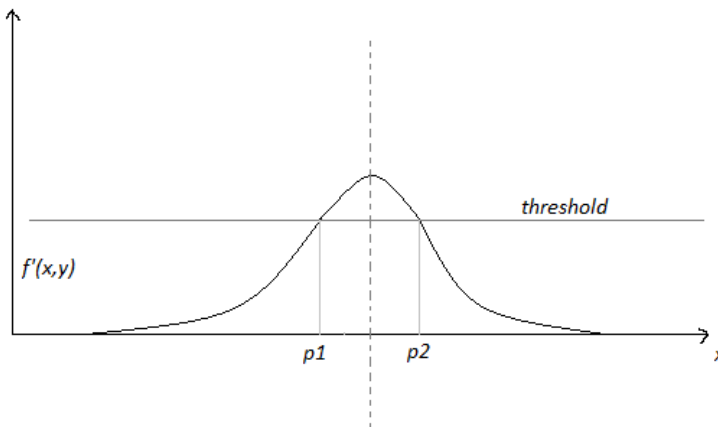


Figure 44: gradient of the curve [1]

A pixel location is declared an edge location if the value of the gradient exceeds some threshold. Edges will have higher pixel intensity values than those surrounding it. So once a threshold is set, we can compare the gradient value to the threshold value and detect an edge whenever the threshold is exceeded [1].

Second way is to find the second order derivative of the smoothed curve. The edge location is at the zero crossing point of the curve as shown in Figure 45:

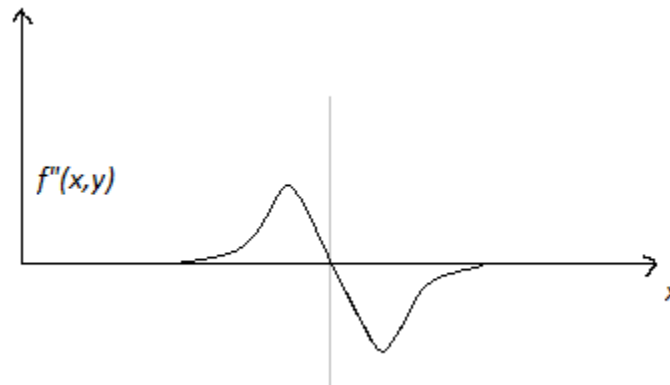


Figure 45: edge location [24]

The first type of detector is called a Gradient based edge detector, and the second type is called a Laplacian based edge detector.

Now, we can use these two types of edge detection operators that are sensitive to this gradual change. The gradual change in intensity causes problems of false edge detection, missing true edges, edge localization, high computational time, problems due to noise etc. [27]. Hence, Canny proposed a new approach to edge detection that is optimal for step edges contaminated by white noise. The optimality of the detector is related to three criteria [25]:

1) The detection criterion should be such that important edges should not be missed and that there should be no spurious responses.

2) The localization criterion should be such that the distance between the actual and located position of the edge should be minimal.

3) The multiple responses to a single edge should be minimized, through the one response criterion.

Thus, convolving an image with a symmetric 2D Gaussian and then differentiating in the direction of the gradient forms a simple and effective directional operator, which meets the three criteria mentioned above [25]. If we define direction  $n$  as perpendicular to the edge direction, the edge location is then at the local maximum of first derivative of  $f(x,y)$  in the direction  $n$ , where  $f(x,y)$  is the Gaussian smoothed image (refer figure 43). Mathematically, the edge location is the zero-crossing point of second derivative of  $f(x,y)$  (i.e.  $f''(x,y)$ ; refer to Figure 45). Canny edge detection is performed in four stages:

1) Gaussian smoothing: The first step is to filter out any noise in the original image before trying to locate and detect any edges. In this stage, the input image is convolved with a symmetric Gaussian mask. The larger the width of the Gaussian mask, the lower is the detector's sensitivity to noise. We will be using a standard 5 X 5 Gaussian mask.

2) Gradient detection: After smoothing the image and eliminating the noise, the next step is to find the edge strength by taking the gradient of the image. The Sobel operator performs a 2-D spatial gradient measurement on an image. The Sobel operator uses a pair of 3x3 convolution masks, one estimating the gradient in the x-direction (columns) and the other estimating the gradient in the y-direction (rows). We can also use two pairs of Prewitt operators, rather than using the Sobel operators. This has an advantage as it causes reduction in required hardware for implementing them. The following are Sobel operators:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \quad G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

The following are Prewitt operators:

$$G_y = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix} \quad G_x = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix}$$

The magnitude of the gradient is then:

$$|G| = |G_x| + |G_y|$$

If we use Sobel operators, we will have to provide multipliers within our hardware to calculate the product of the pixel value times 2, an extra adder to add twice, or a shifter to perform a binary multiply. Also, we require the direction of the gradient for the next step. This is calculated as

$$\theta = \tan^{-1} \frac{G_y}{G_x}$$

These calculations make Sobel operator difficult to implement it on hardware and costly in terms of resource usage. Hence we use the following operator pairs:

$$E_v = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix} \quad E_H = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix} \quad E_{DR} = \begin{bmatrix} 0 & +1 & +1 \\ -1 & 0 & +1 \\ -1 & -1 & 0 \end{bmatrix} \quad E_{DL} = \begin{bmatrix} -1 & -1 & 0 \\ -1 & 0 & +1 \\ 0 & +1 & +1 \end{bmatrix}$$

We, can see the convolution will not involve any multiplication, only subtraction and addition operations. Also, the direction need not be calculated, as the direction of the gradient is assumed to be the direction of the maximum value of the all four calculated magnitudes.

3) Non Maximum Suppression (NMS): This stage involves finding the local maxima in the direction perpendicular to the edge. That is retaining the pixel that has maximum value among its neighbors along the direction of its gradient.

4) Hysteresis: The ‘streaking’ caused in the image from the NMS stage is eliminated using a threshold in this stage. Streaking is the breaking up of an edge contour caused by the operator output fluctuating above and below the threshold [27 & 24]. This stage involves double thresholding. We use 2 thresholds, a high and a low. Any pixel in the image that has a value greater than  $T_1$  is presumed to be an edge pixel, and is marked as such immediately. Then, any pixels that are connected to this edge pixel and that have a value greater than  $T_2$  are also selected as edge pixels. We have, however, modified the double thresholding stage in our implementation. This step makes, Canny edge detection, a very efficient edge detection algorithm. The selection of threshold values, is a major issue. Reference [25] proposes, a self-adjusting threshold algorithm and describes its hardware implementation. However, we will work only with a single threshold value. This is simply to reduce, one stage in the hardware implementation. Thus, our edge detection implementation will not provide true Canny edge detection, however, our simulation results have given decent accuracy. Ultimately, our aim is to feed the edge detected image to our Hough transform module and by adjusting threshold values of the Hough transform voting process, we can easily detect the pillar and the floor edges, even if the edge detection is suboptimal.

Now, we can go into the details of our hardware implementation. Our Canny edge detection module contains hierarchical blocks as shown in Figure 46:

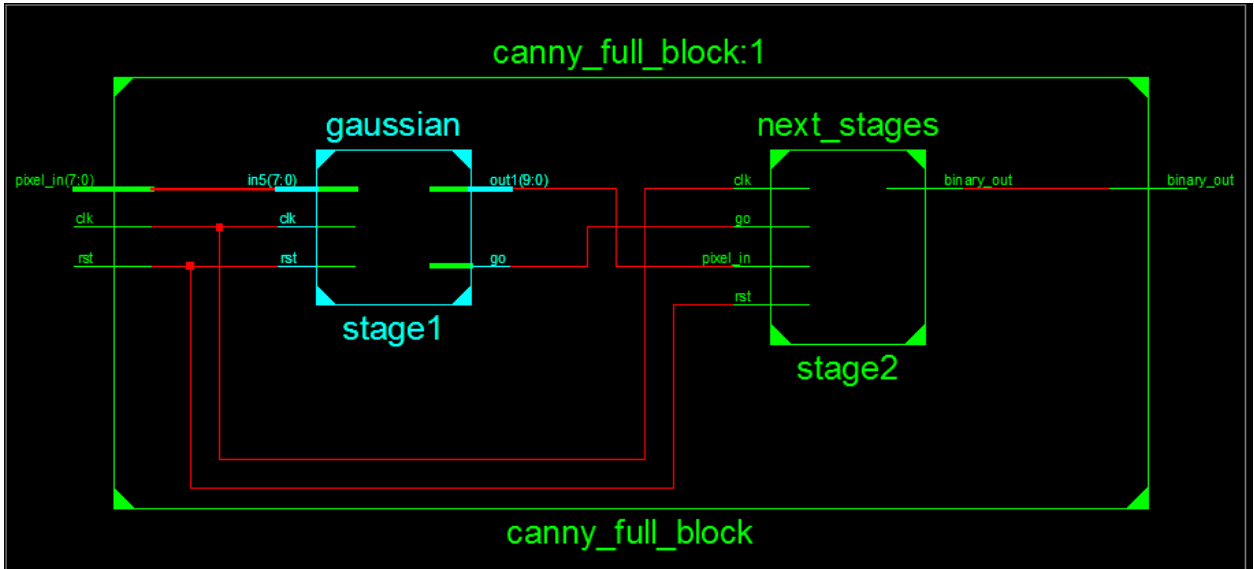


Figure 46: Gaussian stage and the next stages

The first stage performs the Gaussian smoothing and its output is fed to the next stages which contains hierarchical blocks as shown in Figure 47:

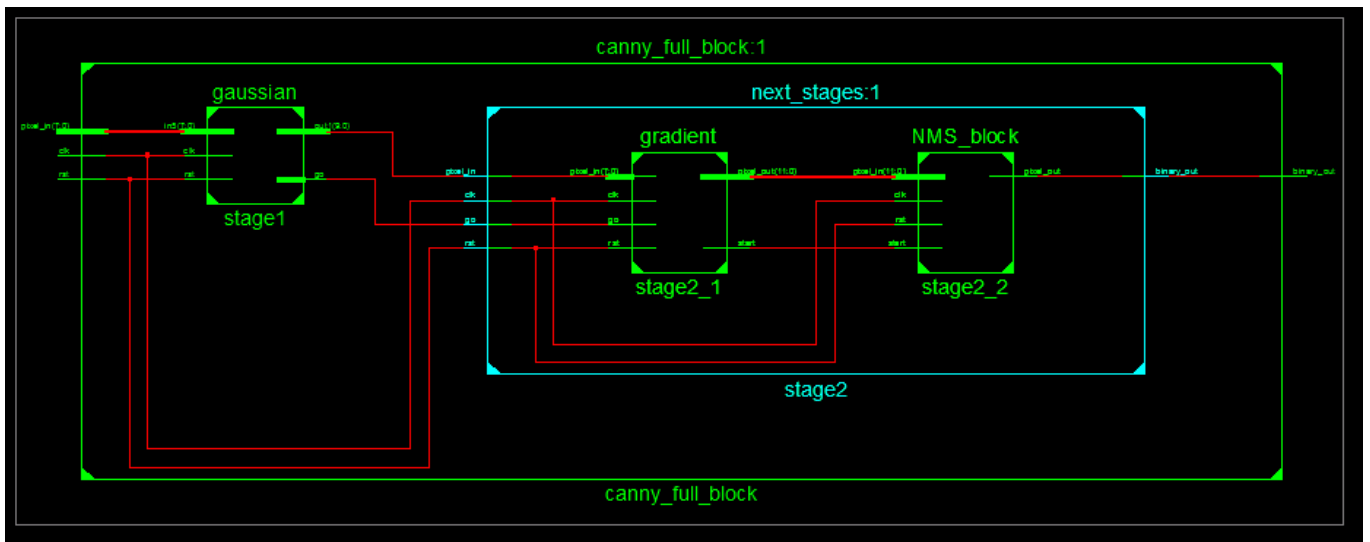


Figure 47: Inner hierarchy of next stages

Now we go into the details of the Gaussian smoothing module. Its block diagram is as shown in Figure 48:

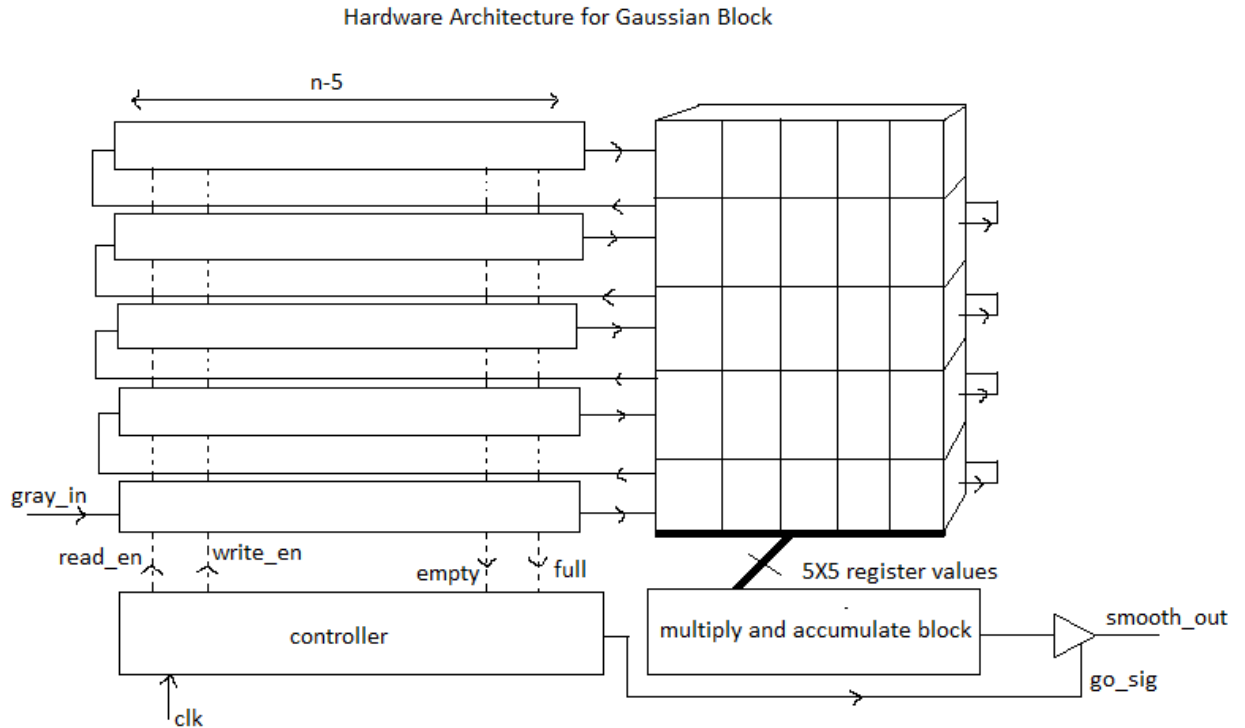


Figure 48: Gaussian stage

The hardware consists of 5 FIFO pipelines. The length of each pipeline is  $n-5$ , for an  $n \times n$  sized image. There is a 25-register block, which is connected to the FIFO as shown in the block diagram of Figure 48. This register block contains the pixels which are to be convolved with the 5x5, Gaussian mask. As pixels shift every clock cycle, it emulates the sliding of the mask over the image. This implementation will give distorted output on the edges of the input image, however, we can eliminate this by padding the image with 0 valued pixels before putting it in the pipeline. The entire 25-register block is connected to the multiply and accumulate block which performs the convolution, and the resultant smoothed pixels are fed to the next stages. The controller, is basically a counter that ensures that the output feeding begins only when the first

pixel reaches center register of the 25-register block. Since the pipeline registers must be filled, for the first output we must wait for  $[(n*3)-2]$  clock cycles, after which one pixel exits the pipeline every clock cycle. Also, the controller ensures, that after the entire  $n \times n$  image has been processed, to the counter is reset for the next image. The controller, also, enables and disables reading and writing to the FIFO, which does not have any purpose in real-time but is used for off-line validation. The hardware implementation of the gradient calculation stage is as shown in the block diagram shown in Figure 49. The pipelined implementation is similar, the primary difference being a 9-register block instead of 25-register block. However, the details of the combinational block will greatly differ from the earlier stage. The 9 register values entering the block have to be convoluted with four operators as described earlier and we have a dedicated sub-block for each of them. Each block represents a direction, Vertical, Horizontal, Left diagonal, Right diagonal. As described earlier, we have only addition and subtraction operations and no multiplications within it.

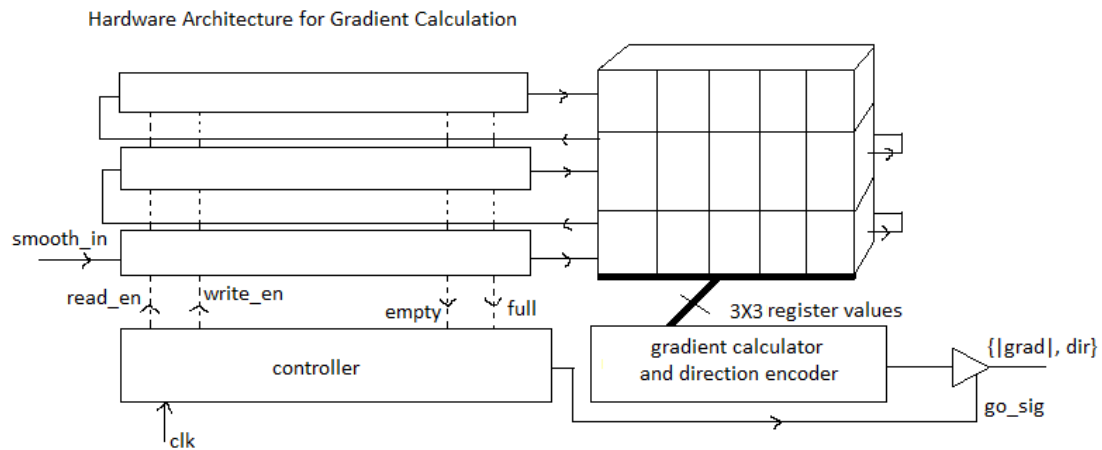


Figure 49: Gradient calculation stage

The combinational block is as shown in Figure 50:

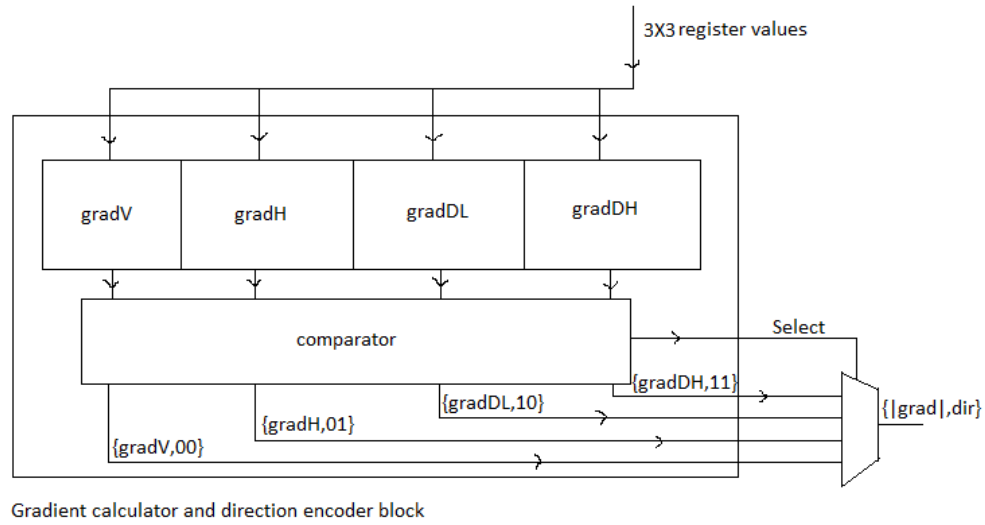


Figure 50: Direction encoder

The absolute values calculated by each of the four blocks are then compared, and the largest value is taken as the gradient value of that particular pixel. The direction of the gradient is the direction of block with the largest magnitude. This direction is encoded as four possible values, 00, 01, 10, 11. These two bits are concatenated to the magnitude value and are fed to the next stage. Thus, next stage which requires the direction values for its operation, will extract the trailing two bits to decode the direction.

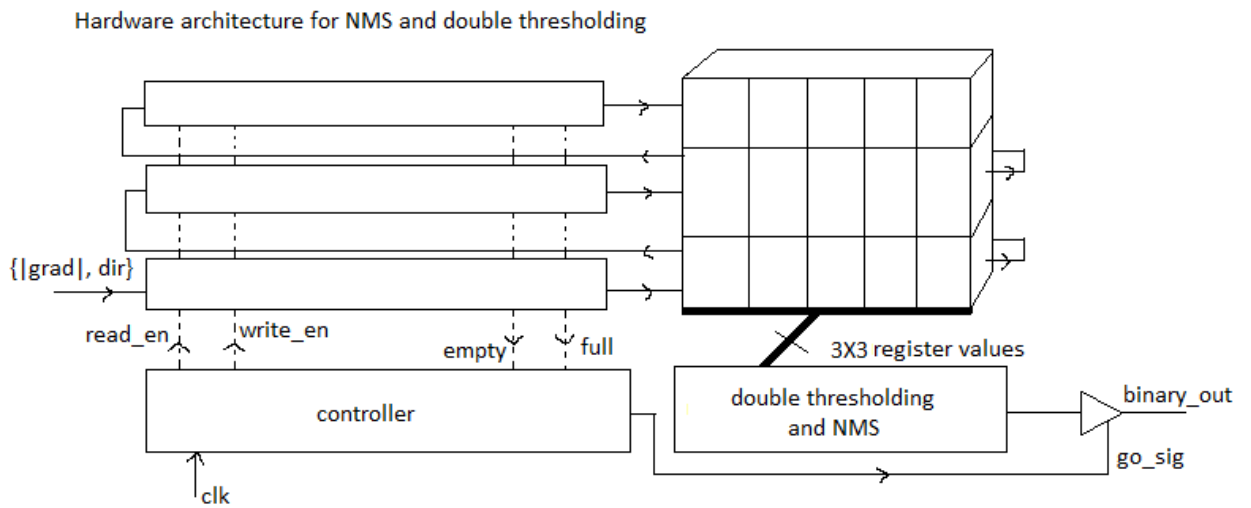


Figure 51: NMS and double thresholding stage



The hardware architecture for the Non-Maximum Suppression (NMS) stage is as shown in Figure 51. Again in this case, we have a 9-register block and a pipelining mechanism that is same as the earlier stages. However, there are significant changes in the combinational block. The combinational block for this stage performs thresholding and NMS. As mentioned earlier, the last two bits of the input values represent the direction of the gradient of that pixel.

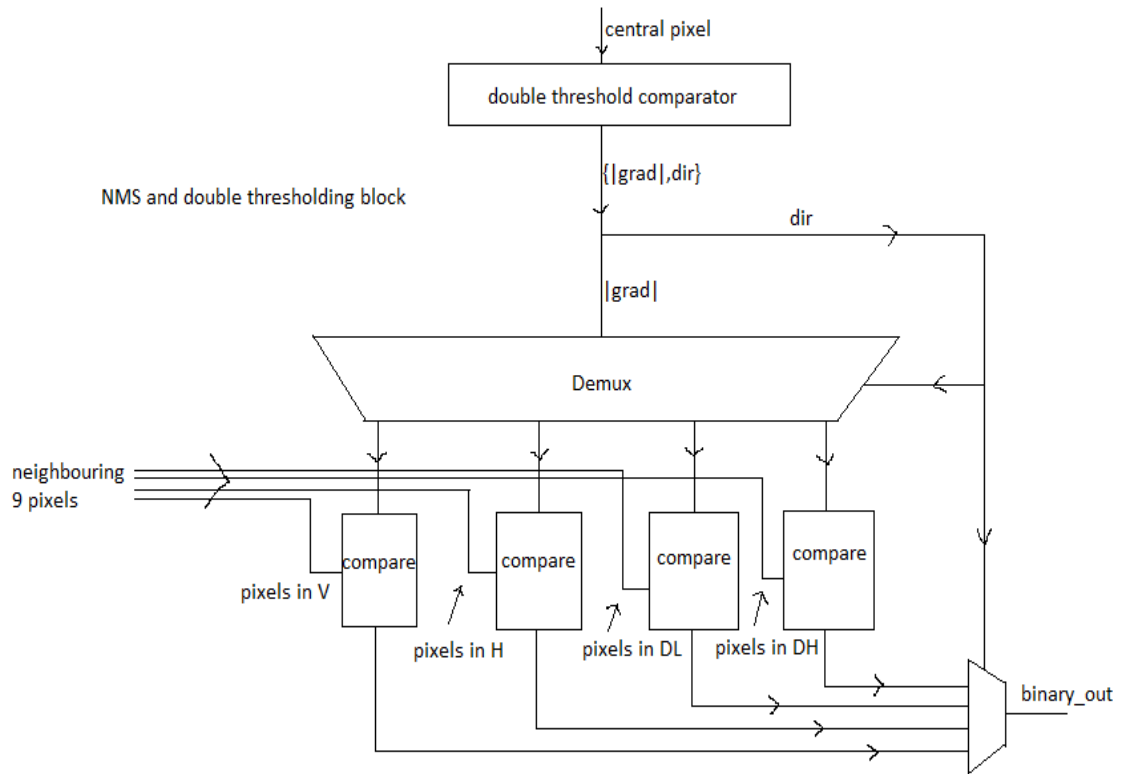


Figure 52: NMS calculation

The mechanism for performing NMS is as follows: The center pixel gradient is compared with the neighboring two pixel gradient values, in the direction represented by the two trailing bits, and if the center pixel gradient value is the highest (i.e. it is a local maxima), then that pixel is retained (i.e. it is assigned value 1), else it is eliminated (i.e. assigned value 0).

The center pixel value is first checked for its threshold value and if it is greater than the threshold, it is forwarded to the NMS blocks, else it is ignored. Then, depending on the two direction bits, the pixel is de-multiplexed to the correct comparator block, where it is compared with its neighboring two pixels. The binary value  $I/O$  is the final output, which is taken from the selected line by a multiplexer which has the two direction bits as its select lines.

All the stages are connected in the top module. Another important thing is the synchronization between each of the stages. The second stage cannot begin filling its FIFO, until the correct output is available from the earlier stage. Thus, the controllers of each stage communicate with each other through the *start* signal.

The *start* signal is issued by the first stage's controller when the first valid output is calculated. In response to this, the next stage begins its counting, and at the same time issues *read enable* to its FIFO. Similar synchronization takes place between the second and third stages. When, the counter overflows, the internal signals of each controller reset the counter, and de-assert the *start* signal. This de-assertion takes place out of sync.

Thus, each counter for the three stages counts independently and also resets independently, but the counting begins only at the assertion of 'start' signal by the previous blocks.

We have synthesized the design. The resource utilization report is as shown in Figure 53.

```

=====
Advanced HDL Synthesis Report

Macro Statistics
# RAMs                                     : 1
  17690x8-bit single-port block Read Only RAM : 1
# MACs                                     : 4
  3x1-to-18-bit MAC                       : 1
  8x4-to-19-bit MAC                       : 1
  9x4-to-18-bit MAC                       : 2
# Multipliers                             : 1
  20x7-bit multiplier                     : 1
# Adders/Subtractors                      : 26
  1-bit adder                             : 1
  1-bit adder carry in                   : 1
  10-bit adder                           : 8
  10-bit subtractor                      : 4
  9-bit subtractor                       : 12
# Adder Trees                             : 5
  18-bit / 2-inputs adder tree           : 1
  8-bit / 8-inputs adder tree            : 1
  9-bit / 4-inputs adder tree            : 3
# Counters                                 : 3
  20-bit up counter                      : 1
  21-bit up counter                      : 2
# Registers                               : 539
  Flip-Flops                             : 539
# Comparators                             : 13
  10-bit comparator greater              : 13
# Multiplexers                            : 17
  1-bit 2-to-1 multiplexer               : 10
  10-bit 2-to-1 multiplexer              : 6
  12-bit 2-to-1 multiplexer              : 1
=====

```

Figure 53: Resource utilization for Canny edge detection

Next we have performed post-synthesis simulation to verify the functionality of the design. We begin with generating input file Using MATLAB. The code for it is as follows:

```

I=imread('peppers.png');
B = imresize(I, [133 133]);
BW=rgb2gray(B);
fid=fopen('C:\Users\srvyas\Desktop\pics\smooth4.txt','w');
for x=1:133
for y=1:133
i= BW(x,y);
il=dec2bin(i,8);
fprintf(fid, '%s \r\n', il);
count=count+1;
end
end
fclose(fid);
imshow(BW);

```

The input gray scale image is as shown in figure 54:



Figure 54: Input gray scale image

The original design has a *start* signal which, when issued, begins the operation of the Canny edge detection module. However, we will be using Verilog task ‘\$readmemb’ to synthesize on-chip RAM, in which we will load our input file and the first block (i.e. Gaussian block) will begin reading this memory as soon as clocking begins. The clock ticking is set from the test bench and the test bench also collects the output binary values using File I/O tasks.

We read the output file using MATLAB code and display the edge detected image. We cannot perform conditional File I/O in Xilinx ISE, hence, we will get unwanted pixels from the simulation and the image will be out of order. Hence, we ‘circshift’ function in MATLAB to fit the binary values in the matrix accurately. The code for it is as shown below:

```
fileID = fopen('C:\Users\srvyas\Desktop\pics\pepppers_out1.txt','r');
A = fscanf(fileID,'%d');
[matp10] = vec2mat(A,133);
myfilter = fspecial('gaussian',[5 5], 1.5);
I=imread('peppers.png');
B = imresize(I, [133 133]);
BW=rgb2gray(B);
imshow(BW);
myfilteredimage = imfilter(BW, myfilter);
Y = circshift(matp10,[0 -30]);
figure();
imshow(Y);
```

The resultant edge detected image is as shown in Figure 55:



Figure 55: edge detected image

We have taken some more results for further validation of the design as shown in Figures 56-58. Compared to the earlier case we have narrowed the two threshold limits (upper and lower threshold): Accordingly we require a higher level of contrast between the two surfaces, for the edge to be detected. As it can be seen from the Figure 56, edge detection is very accurate, because in the gray scale image contrast at the edges is too high. However, in the Figure 57, the part of the V-shaped truss on the right has its gray scale intensity close to that of its background, hence the edge has not been fully detected. Similarly, in the Figure 58, the bright white part at the center has been detected accurately. We must keep in mind that, our second stage is the Hough Transform which has its own threshold value for line detection. Thus, we have to fine tune the two threshold values for accurate feature detection.

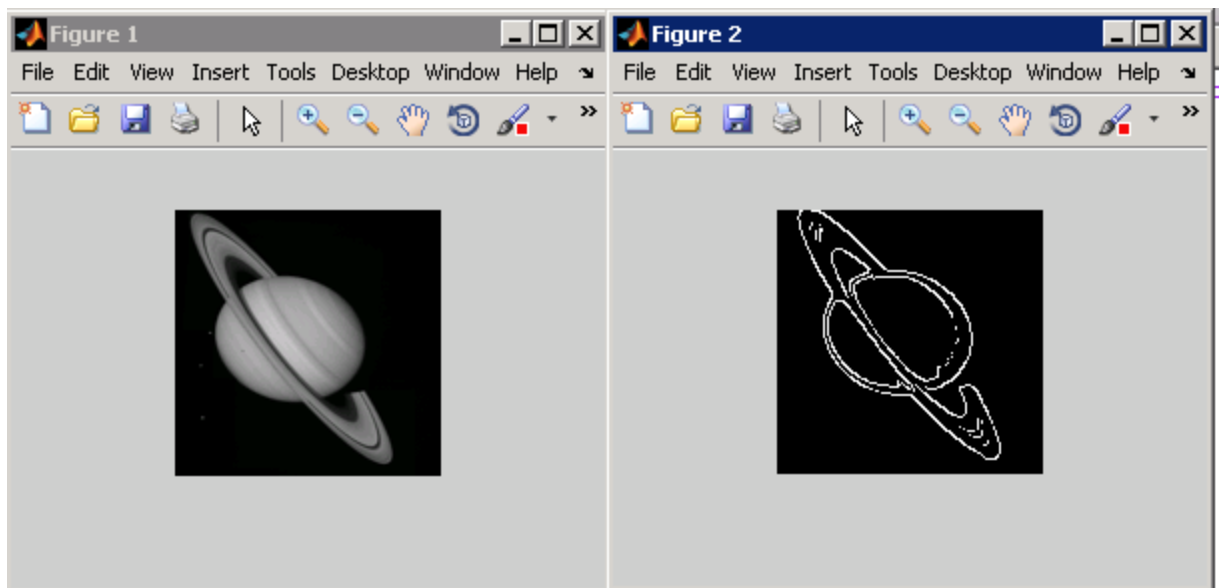


Figure 56: Edge detection result 1.1

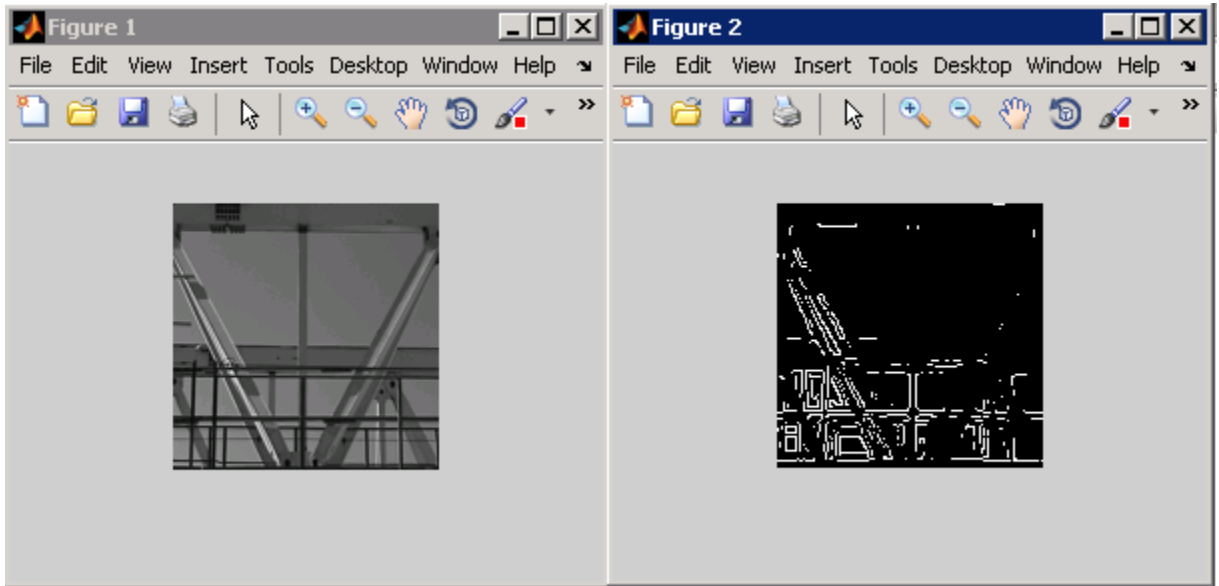


Figure 57: Edge detection result 1.2

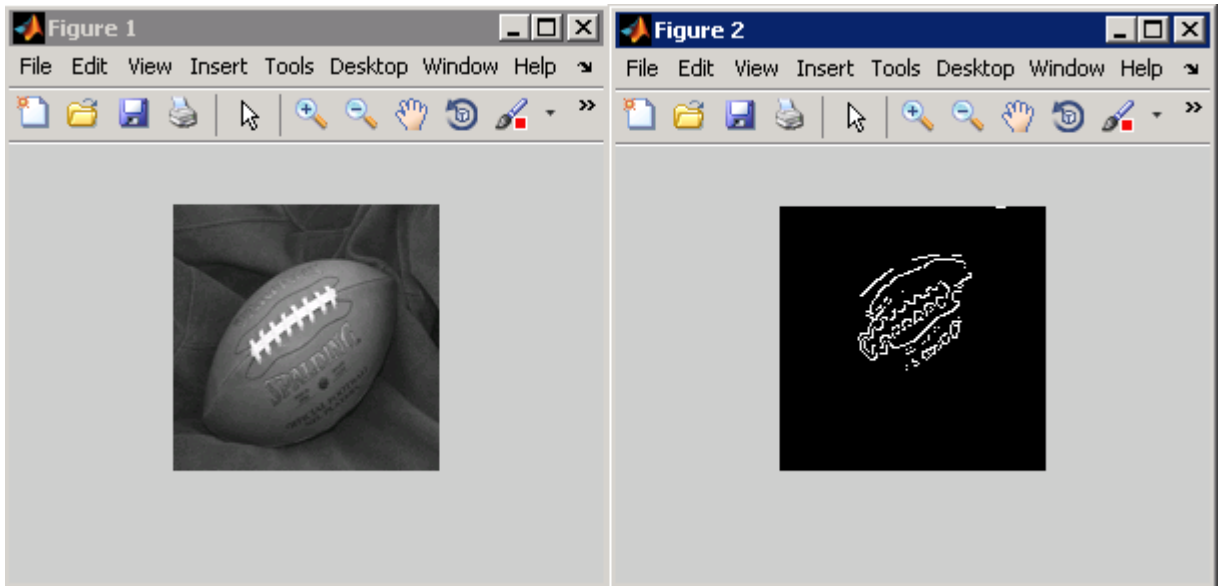


Figure 58: Edge detection result 1.3

In the next section we describe the Hough Transform algorithm in detail and its implementation on an FPGA.

### 3.4 Hough Transform on FPGA

This Hough transform-based feature detection has been performed earlier by us, using C++ code which utilizes the OpenCV Library, which gives the convenience of performing Hough transform on the input image without really going into the details of the Hough transform mechanism. However, it is an objective of this thesis is to implement Hough transform on an FPGA. Hence, in the following subsection we go into the details of how Hough transform actually works, so that we can implement it on an FPGA.

#### Hough Transform Mechanism

The aim of the Hough Transform is to represent regular geometric forms in a parameter space defined by  $\rho$  and  $\theta$ . If a straight line is considered,  $\rho$  represents the normal distance from the origin to the straight-line and  $\theta$  represents the angle between the normal and the X-axis. A straight line, is therefore, represented by the equation:

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

In the parameter space  $\rho$ - $\theta$ , each straight line is represented by a single point as  $A(\rho, \theta)$ . Accordingly, each point that belongs to a straight line has a corresponding sinusoidal curve in the parameter space. That is we can map a line in the  $X$ - $Y$  plane to a point in the  $\rho$ - $\theta$  plane (parameter space) and each point on that line can be mapped to a sinusoid in the parameter space (refer to Figure 59). Similarly, every point in the  $X$ - $Y$  plane will correspond to a sinusoid in the parameter space, that is, if we take a point and plug it into the equation for the parametric form of straight line, and then plot that curve in the parameter space we will obtain a sinusoid. Now, according to the principle of the Hough Transform, all sinusoidal curves each of which corresponds to a point on a straight line, will cross through a single point in the parameter space.

In other words, the input object for the parameter space computation is a contour image. For each feature point in the contour image, the corresponding sinusoidal curve is computed using the relation mentioned above. If a set of feature points in the contour image belongs to the same straight line, their corresponding sinusoidal curves will have a common point in the parameter space. This point is found using a voting process and will comprise the straight line parameters  $\rho$  and  $\theta$ . Using the Hough Transform principle, the task of detecting straight lines in the  $(x, y)$  space, is reduced to determining the intersection points of the sinusoidal curves in the parameter space. These crossing points are simply parameter memory space positions having sufficient votes.

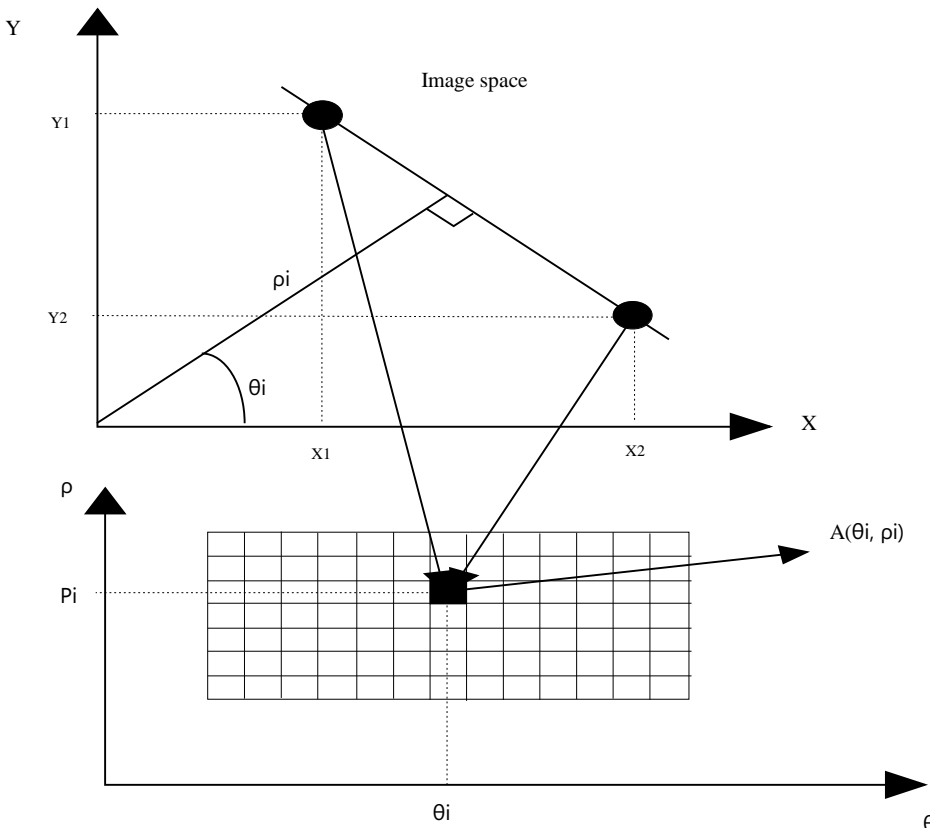


Figure 59: Mapping in parameter space [22]



## Hough Transform Implementation

There have been many efficient Hough transform implementations on FPGAs. In [20] they used a resource efficient implementation based on the concept of incremental Hough transform. The incremental Hough transform algorithm is described in [21]. In [22] they have used the CORDIC algorithm to calculate  $\sin$  and  $\cos$  values. We will be using the implementation described in [23]. This is because, our aim is to beat the software in terms of timing. Hence, we will sacrifice some resource utilization in favor of optimizing timing. Reference [23] describes a pipelined implementation, which is not resource efficient strictly speaking. We have an option of using the pipelined implementation of CORDIC algorithm as described in [22], but the pipelined implementation based on DSP slices described in [23] will be more suited to our case because we are not interested in performing a generic Hough transform on the entire image. We are using Hough transform to detect specific features in our environment such as pillars and floor edges. These features will exist only at specific angles and orientations as we have seen in Sections 2.3 and 2.6. Therefore, we can pre-calculate values of  $\sin$  and  $\cos$  at these specific angles and store them on the FPGA, rather than needing to calculate them in real-time.

The Hough transform is usually calculated with  $\rho$  values positive, and  $\theta$  values ranging from 0 to 360 degrees. Now, if we implement a complete Hough transform (i.e. detecting lines at all angles and orientations) then we would have to calculate  $\cos$  and  $\sin$  from 0 to 360. This would mean having  $360 \times 2$  DSP slices. The Virtex 6 FPGA, which is ideal for Hough transform implementation cannot accommodate this many DSP slices. Hence, a modification for the algorithm is proposed in [23]. The idea is to let the  $\rho$  values be negative and take angles from 0 to 180 degrees. That is, if we have an image of size  $n \times n$ . We assume that  $n \times n$  pixels are

arranged in two dimensional  $xy$ -space such that the origin is in the center of the image. Hence, both coordinates  $x$  and  $y$  take integers in the range  $[-n/2 + 1, n/2]$ . The  $\rho$  values would now range from  $-n/\sqrt{2}$  to  $+n/\sqrt{2}$ . The negative  $\rho$  values would suggest that line is passing above the origin and positive  $\rho$  would suggest that the line is passing below the origin. This would now reduce the DSP slices from 360x2 to 180x2. However, this is still too much. Thus, further reduction in DSP slice requirement can be done using the following algorithm [23]:

```

for i ← 0 to k - 1 do
begin
for θ ← 0 to 89 do
begin
ρ ← xk cos θ + yk sin θ
v[θ][ρ] ← v[θ][ρ] + 1
output (θ, ρ) if v[θ][ρ] = threshold
end
for θ ← 1 to 90 do
begin
ρ ← -x cos (θ) + y sin (θ)
v[180-θ][ρ] = v[180-θ][ρ] + 1
output (θ, ρ) if v[θ][ρ] = threshold
end
end

```

We have to calculate the value of  $\rho$ , for every pixel value  $(x_k, y_k)$ , for  $\theta$  ranging from 0 to 180 degrees. Now, “ $\cos(180-\theta)=-\cos\theta$ ” and “ $\sin(180-\theta)=\sin\theta$ ”. Consider the equation  $\rho = x \cos \theta + y \sin \theta$ , and say for  $\theta=1$ ,  $\cos(179) = -\cos 1$  and  $\sin(179)= \sin 1$ ; we get the equation  $\rho = x \cos 1 + y \sin 1$  for  $\theta=1$  and  $\rho = -x \cos 1 + y \sin 1$  for  $\theta=179$ . Thus, we need not calculate value for both  $\theta=1$  and  $\theta=179$ . We calculate trigonometric ratios only for angles  $0^\circ$  to  $90^\circ$ , the values of  $\rho$  for this range is obtained by adding the values of sine and cosine, and the values  $\rho$  for the angles  $91^\circ$  to  $180^\circ$  can be obtained by subtracting the previously calculated values of sine and cosine. This reduces the hardware requirement (specifically the number of DSP slices) and allows us to perform calculation of two sets of  $\rho$  values in parallel. Thus, we can exploit parallelism in this algorithm and also simultaneously pipeline the entire design to give very high

throughput. Also our DSP slice requirement goes down to 90x2. We first develop a generic Hough transform as implemented in [23]. We will then start making modifications and optimize our hardware design. Also, it is important to note that our target FPGA is a Spartan 6 FPGA and not Virtex 6 (reference [23] uses Virtex 6). The overall design is described in Figure 60:

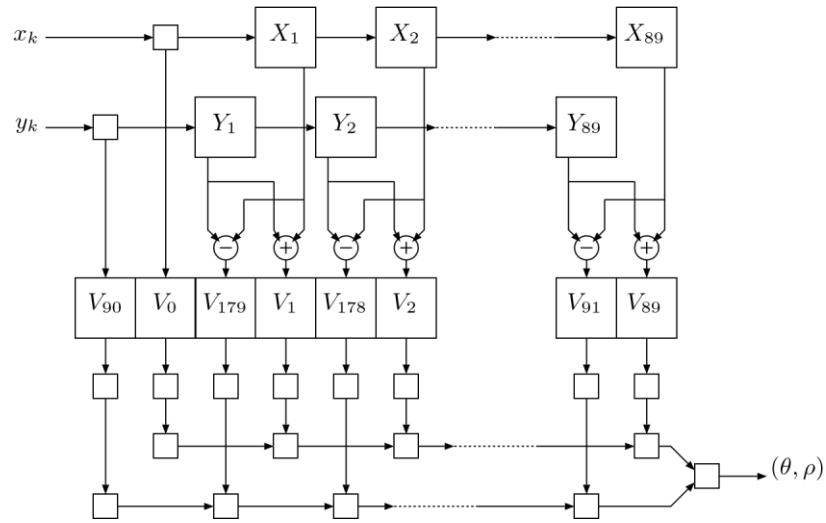


Figure 60: Complete Hough Transform Block Diagram [23]

We will not go into the details of the implementation in [23]. We will directly list the resources utilization for the design in Figure 61:

Adders/Subtractors	: 29995
24-bit subtractor	: 89
25-bit adder	: 29904
8-bit subtractor	: 2
# Registers	: 31158
1-bit register	: 2
11-bit register	: 356
25-bit register	: 30260
8-bit register	: 540
# Comparators	: 90602
11-bit comparator equal	: 178
25-bit comparator greater	: 60698
25-bit comparator lessequal	: 29726
# Multiplexers	: 30260
11-bit 2-to-1 multiplexer	: 178
25-bit 2-to-1 multiplexer	: 29904
8-bit 2-to-1 multiplexer	: 178
# Tristates	: 1440
1-bit tristate buffer	: 1440

Figure 61: Resource Utilization for generic Hough Transform

Now, we will customize this optimized version of Hough transform implementation specifically for the purpose of Pillar detection and Floor edge detection. The first step is to include a quadrant changer module. We will be interfacing the Hough transform module with the Canny edge detection module. The output image from the Canny edge detection will be in a general form where the origin of the image (i.e. *pixel (0,0)*) is at the top left corner of the image. Hence, we need to change the quadrant system and bring the  $(0,0)$  pixel to the center of the image. This has been implemented by passing the incoming pixel's  $X$  and  $Y$  coordinates through a module which calculates new coordinate values using the mapping equations. The implementation of the DSP slices and adder/subtractor system is same as that in [23], however, we have selected only specific angles. The angles selected are on the basis of Hough transform outputs observed in the software implementation for detecting pillar and floor edges. The vertical pillars would always be oriented at an angle of 0 to 3 degrees or 178 to 180 degrees. Also, the floor edges would always exist at angles 43-45 degrees and 133-135 degrees as seen in Figure 62:

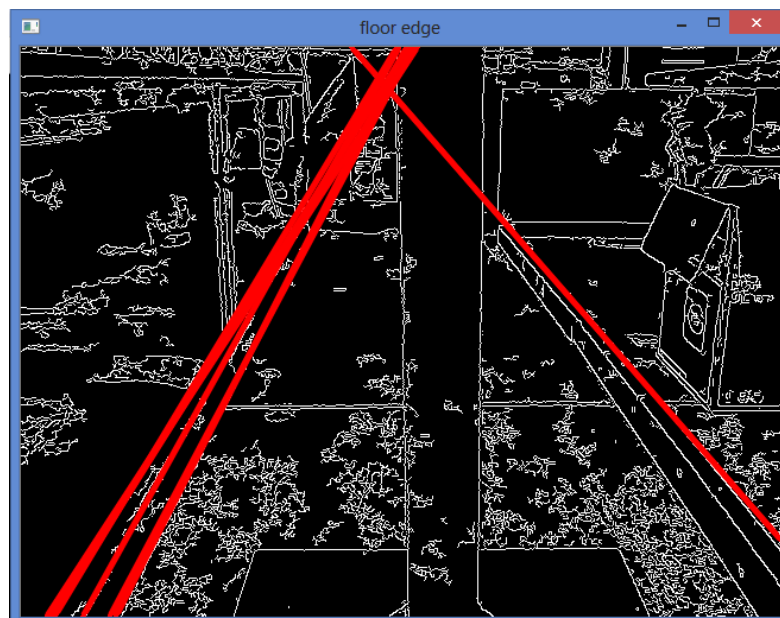


Figure 62: Inclined floor lines

This significantly reduces our DSP slice requirements enabling us to synthesize the design on a Spartan 6 FPGA.

We have implemented the accumulator matrix by synthesizing memory from clocked registers and interfacing it with comparators, as opposed to ROM addressing and the comparator system implemented in [23]. This change is a result of the observation that a ROM-based implementation has timing issues (due to port contention) and therefore is not advisable for the pipelined design.

Our last modification is the parallel shift register system which would provides parallel output values for both pillar and floor edges. That is, we have a four port output, a  $\rho$  and  $\theta$  for pillar and a  $\rho$  and  $\theta$  for floor edge. Thus, one advantage of implementing the Hough transform in hardware, is that we can get parallel output for both floor edges and the pillar, as opposed to software which can only perform these operations sequentially.

In the following subsection we have given the details of each RTL block for the Hough transform module. The black box of the *hough* module looks as shown in the following block diagram shown in Figure 63:

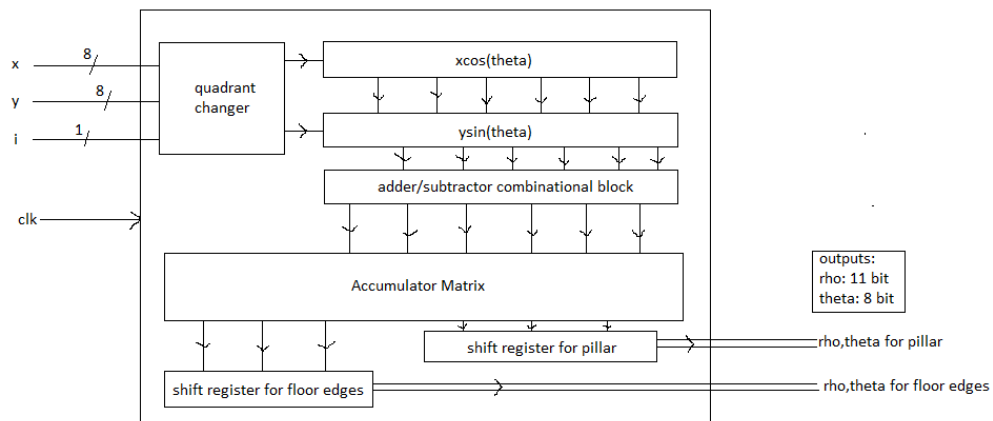


Figure 63: detailed block diagram for Hough Transform

There are three inputs,  $x$ ,  $y$  and  $I$ . The  $x$  and  $y$  inputs are the pixel numbers entering the block and the  $I$  is the intensity of that particular  $(x,y)$  pixel. Since, the input image is the edge detected image, at a pixel that is part of the detected edge, the intensity will be  $I$  and the intensity value at the remaining pixels will be  $0$ . Now, the first block in the hierarchy is the *quadrant\_changer* block. The block is as shown in figure 64:

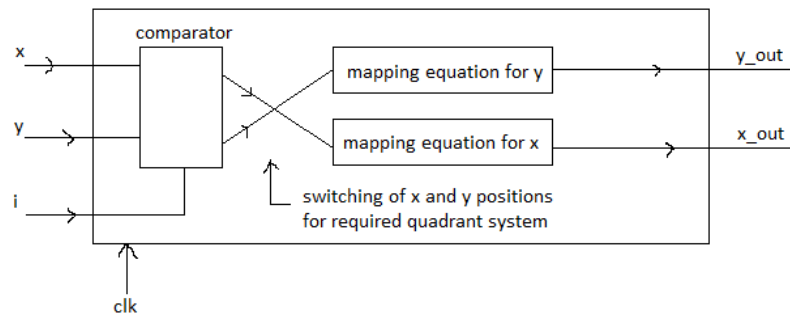


Figure 64: quadrant shifter block

As described earlier we have to change the quadrant system, moving the  $(0,0)$  pixel at the top left corner to the center of the image. Also we have to note, the general convention for denoting a pixel is  $(row, column)$ , however, we will be following a Cartesian co-ordinate system where each pixel is denoted as a point  $(x,y)$  in the Cartesian plane with  $x$  representing the columns and  $y$  representing the rows. Thus, our first block has two responsibilities: the first is to eliminate the pixel values with  $0$  intensity and the second is to map the input image pixels to the desired quadrant system. Thus, we have a comparator that checks the  $I$  value every clock cycle and only allows  $x$ ,  $y$  coordinates with intensity  $I$  to pass through and be stored in registers  $xreg$  and  $yreg$  respectively. These  $x$  and  $y$  coordinates are then mapped to the desired quadrant system using the mapping equations. The mapping equations are actually modeled using a sequential block which is sensitive to the values of two registers  $xreg$  and  $yreg$ . The values on the output

port of this block represent the required values of the pixels as calculated by the mapping equations.

Now, the subsequent blocks are designed to calculate  $x\cos\theta$  and  $y\sin\theta$  from  $\theta$ : 1 to 89 degrees. These calculations are done in parallel and contain the DSP blocks. As mentioned earlier, we have pre calculated the values of  $\cos\theta$  and  $\sin\theta$  and stored them in registers. Let us look into the detail of one multiplication block using DSP slices (refer figure 65).

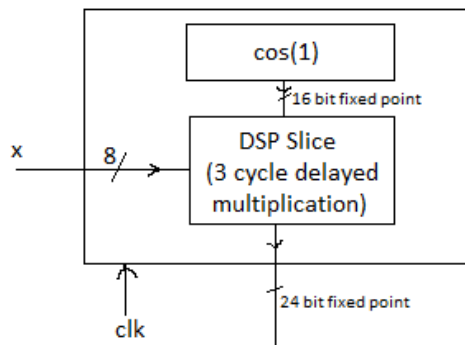


Figure 65: calculation using DSP slices

As we can see the input value is an 8 bit value and the cos value we have stored is a 16 bit fixed point value. The DSP slice will perform the multiplication for these two values and give the output as a 24 bit fixed point value. Now, this forms one multiplication block; we have to cascade multiple such blocks to get a pipelined architecture for calculating  $x\cos\theta$  for all desired values of  $\theta$ . This pipelining is modeled as clock synchronized sequential block. The block to calculate  $y\sin\theta$  is the exact clone of the block to calculate  $x\cos\theta$  with the obvious difference, that we will be storing the pre-calculated 16 bit fixed point  $\sin$  value. Also it is important to note that, DSP slice also has inbuilt pipelined stages and it typically takes 3 clock cycles for the DSP slice to multiply the two input values. Hence, the first output will start arriving from our blocks three cycles after we provide it with inputs, but after the first output arrives, the consequent outputs

will start coming out every consequent clock cycle on account of the pipelined architecture of DSP and our block.

Next we have the combinational adder and subtractors to calculate  $\rho$ . These are modeled by the concurrent *assign* statements.

Now, the prime block of our Hough transform module is the accumulator matrix. This can be modeled in several ways as described in [20], [22] and [23]. However, we will be simplifying the design. We have the liberty to do so, because we are not designing a generic Hough transform. We have chosen only a specific set of angles at which we will be calculating the values of  $\rho$ . Also, in order to reduce the resource utilization, we will be making some assumptions. Our rho values range from  $-n/\sqrt{2}$  to  $+n/\sqrt{2}$ . The negative rho values indicate that lines are passing above the origin. It can be seen in the picture 62, the lines for the pillar as well as the floor edges, pass above the center of the image (above the origin). This gives us an added advantage of taking only the negative values of  $\rho$ . Thus, our accumulator matrix size is drastically reduced. In terms of resource utilization, the reduction in the size of the accumulator matrix has many implications. The accumulator matrix is modeled as multiple clocked registers, with each register corresponding to the integer value of  $\rho$  ranging from 0 to  $-n/\sqrt{2}$ . The incoming calculated fixed point rho values is first compared with the smallest integer  $\rho$  value, if it smaller than the counter in that particular register corresponding to the smallest  $\rho$  value is incremented by one. The counter value is compared with the threshold value. If the counter value is greater than the threshold value, then the integer value of rho indicates a detected line, and that value is given as output. If the incoming calculated fixed point  $\rho$  value is not smaller than the smallest integer stored  $\rho$  value than the next higher stored value is compared and the operation is repeated. This entire operation has to take place in a single clock cycle. This is why, reducing the hardware



utilization is of utmost importance, because with more hardware, the operating frequency of the design would be hindered to great extent. Basically, each operation requires two comparators (one for comparing the  $\rho$  values and another for comparing counter with threshold value), one adder (to increment the counter value) and two clocked registers for storing integer  $\rho$  value and the counter value.

The last block is the series of shift registers which will take the values of  $\rho$  and  $\theta$  for each detected line from the accumulator matrix block and shift them to output. There will be two pairs of shift registers, one for detecting the pillar and other for detecting the floor edges. Each of the shift register is actually a custom designed comparator-based-switch block. The design for it as shown in the diagram of Figure 66.

We need to mention the fact in our design we cannot have multiple assignments to the same wire. That is, an output port of a block cannot have multiple drivers. But, we have a block that is supposed to be cascaded serially and also connected to the output of the accumulator matrix. Our block has to take two inputs one from the accumulator matrix and other from the block cascaded to its inputs.

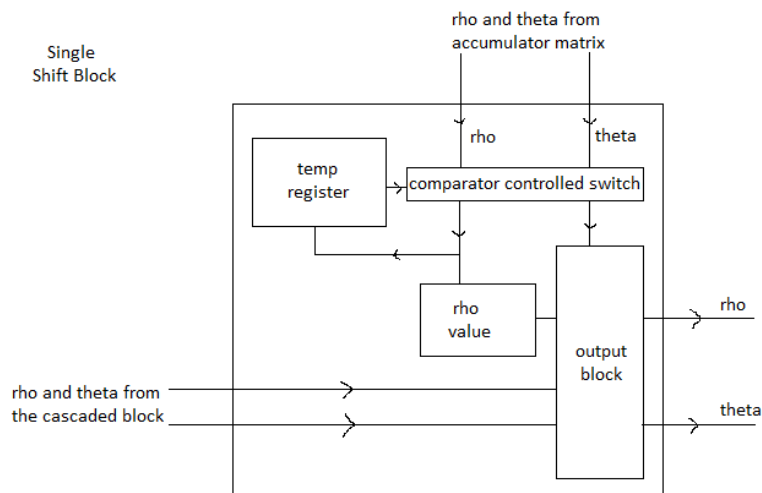


Figure 66: Special unit of shift chain

Now, we are not going to get a continuous output from the accumulator matrix because accumulator matrix will give the output only when the counter value crosses the threshold and that is not going to happen often. However, at the instant when we do get the output from the accumulator matrix we have to make a decision to choose either to take the inputs from the cascaded block or to take the input from the accumulator matrix. In this case the outputs on the block cascaded to the current block's inputs have to be held steady for one clock cycle, and in this clock cycle our current block has to take the value from the accumulator matrix and push it out to the next block. Then, in the next cycle it can take the value from the cascaded block. We have to manipulate registers within the block such that we can give correct output at the required instant. The only situation, in which the value from the cascaded block is lost, is when two consecutive instances of the accumulator blocks (that is lines at two consecutive values of  $\theta$ ) are detected simultaneously. However, this is a very rare and near impossible event (as we have observed from experimental simulations).

The block functions in the following way: When there is a change in the input port connected to the accumulator matrix, following series of events occur. The input value of  $\rho$  is checked against the previously stored value in a temporary register. Obviously for the first time this is going to be true, because the temporary register on a FPGA will be initialized to zero. If the condition is true, which it will be the first time, the output port of this shift block will be assigned the value coming from the accumulator register and also this value will be stored in the temporary register for a future comparison. If the condition is not true then this implies that the  $\rho$  value coming from the accumulator is a duplicate value, that is, a line with that particular  $\rho$  and  $\theta$  value has already been detected previously. Thus, the current value will be discarded and the value from the cascaded shift register block will be assigned to the output ports. But, these events

happen only on change in value of the input ports connected to the accumulator matrix in absence of which the value on the input ports connected to the cascaded block will be directly assigned to the output. Next, we synthesize this design and check its resource utilization. This is as in the table shown in Figure 67:

# Adders/Subtractors	: 14
25-bit adder	: 6
25-bit subtractor	: 6
8-bit subtractor	: 2
# Counters	: 996
25-bit up counter	: 996
# Registers	: 1086
Flip-Flops	: 1086
# Comparators	: 3024
11-bit comparator equal	: 12
25-bit comparator greater	: 3012
# Multiplexers	: 1992
1-bit 2-to-1 multiplexer	: 996
11-bit 2-to-1 multiplexer	: 12
25-bit 2-to-1 multiplexer	: 972
8-bit 2-to-1 multiplexer	: 12

---

Figure 67: Resource utilization for Customized Hough Transform

Also, the maximum operating frequency of our design is 103 MHz. We performed a post synthesis simulation using Xilinx ISE. First we take the image from the software implementation, as shown in Figure 68:

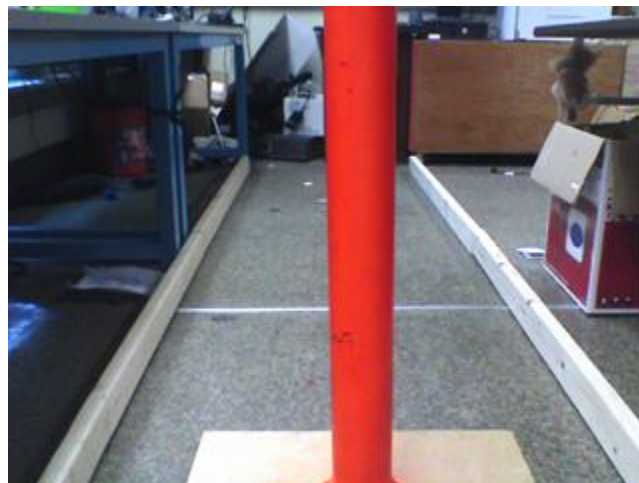


Figure 68: Original image before edge detection is performed

Then we wrote a script in MATLAB as shown below to generate the edge detected pixel

list for this image and stored it in a file.

```
I=imread('C:\Users\srvyas\Desktop\pics\pillar.gif');
BW=edge(I);
fid=fopen('C:\Users\srvyas\Desktop\pics\inp11.txt','w');
for x=1:519
    for y=1:654
        xval=x;
        x1=dec2bin(x,8);
        fprintf(fid, '%s \r\n', x1);
        count=count+1;
        yval=y;
        y1=dec2bin(y,8);
        fprintf(fid, '%s \r\n', y1);
        count=count+1;
        i=BW(xval,yval);
        i1=dec2bin(i,1);
        fprintf(fid, '%s \r\n', i1);
        if (i==0)
            count=count+1;
        end
    end
    count3=count3+1;
end
fclose(fid);
imshow(BW);
```

This would be the input file containing the pixel coordinates and the intensity value. The edge detected image and the file are as shown in Figure 69 and Figure 70 respectively:



Figure 69: Edge detection performed on the original image

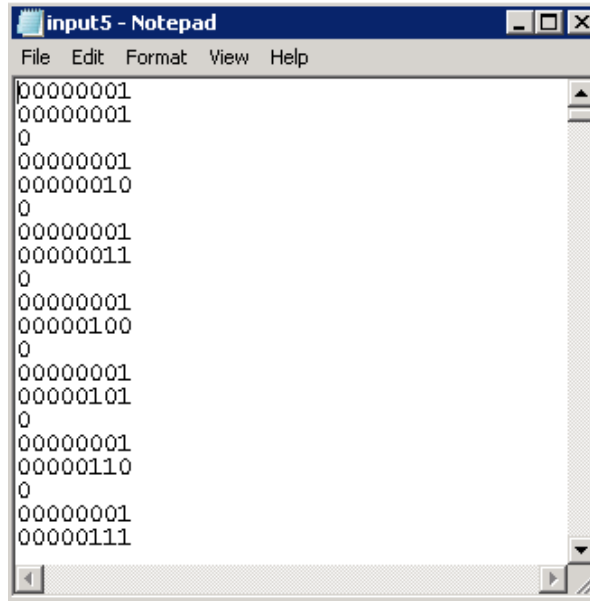


Figure 70: Input text file

The MATLAB also treats the image with pixel co-ordinate  $(0,0)$  at the top left corner and hence, the file generated by MATLAB would be highly appropriate to test our design (i.e. validate our quadrant changing mechanism). The Output waveform for the simulation is as in Figure 71.  $xx$  and  $yy$  are 8-bit input ports that take pixel values from  $(0,0)$  to  $(120,120)$  in Raster scanning order: 00, 01, 02.... In the output waveform below,  $(16,102)$  to  $(16,115)$  is the current input. Port  $ii$  is a 1-bit input port that takes in value of the intensity level  $I$  or  $0$  at that pixel value (the input image is the binary edge detected image). In the waveform for example at pixel value  $(16,104)$  there is an edge pixel and hence the intensity is  $1$ .  $\rho\_out1$  and  $\theta\_out1$  output ports take the rho-theta value of detected pillar edges and  $\rho\_out2$  and  $\theta\_out2$  take it for detected floor edges.

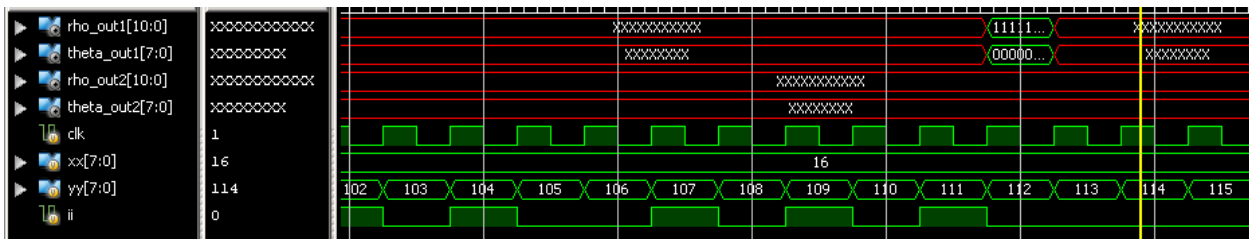


Figure 71: Output waveform

We get two output files containing the output  $\rho$  and  $\theta$  values for pillar and for the floor edges. We note down the simulation timing for  $512 \times 512$  image. It takes 3.33ns at 100MHZ to calculate  $\rho$  and  $\theta$  values.

Now we use timestamps in our software to calculate the time taken by the software to process the Hough transform function. The stamps observed in various iterations give the values ranging from 5ms to as high as 17 ms. We can observe that our hardware implementation beats the software even at much lower operating frequency of 100MHZ.

### 3.5 Timing Results

We will compare the timing results in software (MS visual studio) with that in hardware (Xilinx ISIM simulation software) for a  $120 \times 120$  image. The C++ code is run at the local processor speed somewhere at 1GHZ. The simulations in ISM are done at the frequency of 100 MHZ (maximum operating frequency: 103.916 MHZ, clock period of 9.623ns). The results are as shown in Figure 72:

Performance in Software	
Function	timing
Canny edge detection	1-3 ms
Hough Transform	7-8 ms

Figure 72: timing performance of software implementation<sup>6</sup>

---

<sup>6</sup> The input image for Canny edge detection OpenCV function is a 8-bit single channel Gray scale image, which is same as that used for FPGA implementation. Also, the OpenCV implementation uses an approximate gradient calculation method rather than high precision calculation to have a fair comparison with the FPGA implementation.

On the FPGAs, computation of timing only for the first frame matters, as this is the time required to fill all the buffers in Canny module and ripple through the DSP slices in Hough transform module. The remaining frames will be processed immediately following the first frame as the buffers are already filled and DSP slices have all three pipeline stages filled.

Performance in Hardware	
Function	timing
Canny edge detection	0.36ms
Hough Transform	0.144ms

Figure 73: timing performance of hardware implementation<sup>7</sup>

The total time taken by Canny edge detection function and the Hough Transform function in Visual studio, after resizing operation is 11 to 14 ms for the first frame. The same operations take a total of 0.50767 ms in the FPGA implementation, excluding the time taken for storing the output from Canny edge detection and subsequent extraction by the Hough Transform module.

---

<sup>7</sup> The  $\rho$  value obtained from the DSP slice is 25 bit (11bit.14bit) *fixed point*, giving a good output precision. The output from OpenCV is not *fixed point* but of *double* value, hence, it is capable of producing much higher precision output. Thus, timing comparison for  $\rho$  value calculations may not be entirely fair, however, remaining algorithm is just counter incrementing and Threshold comparison. Hence, this comparison positively gives us a decent idea of overall speed-up that is achieved through FPGAs.

## Chapter 4

### Conclusion

It is a fact that, a direct implementation of an algorithm in hardware can achieve higher levels of parallelism than a microprocessor based design and are several magnitude faster, use less area and have lower power consumption [28]. We have proved this fact to some extent: Our implementation of Hough Transform and Canny Edge detection algorithms on FPGA runs many times faster as compared to the software based approach. Also, theory of FPGAs consuming lower power as compared to the microprocessors is based on the concept of dynamic power dissipation within the silicon fabric, which is directly proportional to the operating frequency. That is, higher switching rate of transistor on a semiconductor fabric (therefore higher clock frequency of the device) higher is the power dissipation (therefore higher power consumption of the device). This is mathematically represented by the formula:

$$p = cv^2f$$

Where  $p$  is dynamic power dissipation,  $c$  is the load capacitance and  $f$  is the clock frequency. The maximum operating frequency of our FPGA implementation was 100MHZ, which is much lower as compared to any sophisticated processor which runs in the GHZ range. The lower operating frequency for FPGA corresponds to much lower power dissipation (assuming the FPGA and microprocessors both operate at almost same voltage of 5V).

Thus FPGAs are well suited to applications in the field of robotics and automation. It provides optimization in terms of timing performance, which facilitates robust implementations



of complicated algorithms and at the same time provides optimization in terms of power consumption which is crucial to mobile robotic platforms.

Our Canny edge detection implementation gives decent results but still consumes a lot hardware resources. An interesting technique of array computing may help address this issue. This technique works on a unique architecture, where the entire algorithm is decomposed into smaller sub-algorithms, each of which actually generates a partial result and finally the combining all the partial results, the solution to the main problem is achieved. Basically, the architecture consists several hardware units, each capable of communicating with its immediate neighbors and each of which processes and generates a partial results based upon the input received from one of its neighbors and the result generated is passed on to another neighbor.

We used Hough transform (and line based SLAM) in this thesis, because the objects found in the indoor environment can be characterized by line-based edges like pillars. However, there are several possible features found in the indoor environment which can be characterized by shorter line segments like legs of chairs and tables. In such a case, we can use incremental Hough Transform for detection of shorter segments as described in [21]. This incremental Hough Transform can be implemented on FPGA, in resource efficient manner as described in [20].

We have restricted the implementation of Monocular SLAM to indoor environment. However, as we proceed towards developing a non-generic SLAM, capable of operating in all possible environments we have to avoid using line based SLAM. Line-based edges are the primary characteristic only of the man-made objects found in an indoor environment, but this is not the case in natural world. Features such as trees, rocks, mounds are not characterized by straight lines. In such a case we have to rely on corner point detection. As we have mentioned several times earlier, majority of the SLAM implementations are based on Harris corner

detection. During our literature review, we have come across several such papers, one of which is [29].

We have also described the camera model in our thesis, as a constant velocity model. This model assumes a constant linear and angular velocity of the camera corrupted by noise. This noise is the product of unknown linear and angular acceleration (represented by zero mean Gaussian process) and time impulse ( $\Delta t$ ). This model prevents the use of High speed camera because sudden jerk motions that complement a fast moving robotic platform cannot be modeled by our current camera model. [30] Gives a higher order extension to the constant velocity model. In this new model velocity is no more constant (it was not really constant, there was always an addition of noise) but can vary over a time impulse and is represented by kinematic equation:  $u = v + at$ . Also, in this model, linear and angular acceleration is constant but corrupted with noise. This noise is the product of unknown linear and angular jitter (represented by zero mean Gaussian process) and time impulse ( $\Delta t$ ). Such a model can now be used in Monocular SLAM implementations which makes use of fast moving robotic platforms.

Finally, we conclude with an assertion that combination of Microprocessors and ASICs interfaced with FPGA fabric, forms the best computing unit that can be a part of an ideal embedded system for implementation of complex systems having wide range of application in the contemporary world.

# BIBLIOGRAPHY

- [1] Digital Image Processing by Gonzalez, Rafael C./ Woods, Richard E
- [2] OReilly-LearningOpenCV
- [3] OpenCV official online tutorial
- [4] A Solution to the Simultaneous Localisation and Map Building (SLAM) Problem,  
M.W.M.G. Dissanayake, P.Newman, S. Clark, H.F. Durrant-Whyte and M. Csorba.
- [5] Monocular Visual SLAM based on Inverse Depth Parametrization, Mälardalen Univeristy  
School of Inovation, Design and Engineering Author: Víctor Rivero
- [6] Dimensionless Monocular SLAM, Javier Civera<sup>1</sup>, Andrew J. Davison, and J. M. M.  
Montiel
- [7] MonoSLAM: Real-Time Single Camera SLAM, Andrew J. Davison, Ian D. Reid, Member,  
IEEE, Nicholas D. Molton, and Olivier Stasse, Member, IEEE
- [8] Kalman and Extended Kalman Filters: Concept, Derivation and Properties, Maria Isabel  
Ribeiro Institute for Systems and Robotics Instituto Superior T´ecnico
- [9] An Introduction to the Kalman Filter by Greg Welch and Gary Bishop
- [10] SLAM with Corner Features Based on a Relative Map, Manuel Altermatt, Agostino  
Martinelli, Nicola Tomatis and Roland Siegwart
- [11] Undelayed Initialization in Bearing Only SLAM, Joan Sola, Andre Monin, Michel Devy and  
Thomas Lemaire
- [12] Constrained Initialization for Bearing-Only SLAM, Tim Bailey
- [13] Delayed Feature Initialization for Inverse Depth Monocular SLAM, Rodrigo Munguia and  
Antoni Grau

- [14] A practical 3D Bearing-Only SLAM algorithm, Thomas Lemaire, Simon Lacroix and Joan Sola
- [15] Building a Partial 3D Line-based Map using a Monocular SLAM, Guoxuan Zhang and Il Hong Suh, *senior member*, IEEE.
- [16] Straight-lines modelling using planar information for monocular slam, Andr e m. Santana, Adelardo a.d. Medeiros
- [17] Monocular visual mapping with the Fast Hough Transform, Nicolau Leal, Werneck and Anna Helena Reali Costa
- [18] A line feature based SLAM with low grade range sensors using geometric constraints and active exploration for mobile robot, Young-Ho Choi·Tae-Kyeong Lee·Se-Young Oh
- [19] Scan Matching in the Hough Domain, Andrea Censi, Luca Iocchi and Giorgio Grisetti
- [20] Implementation of Hough Transform Using Resource Efficient FPGA Architecture, S.Subbiah, A.Regal
- [21] Incremental local Hough Transform for line segment extraction, Rui F. C. Guerreiro, Pedro M. Q. Aguiar
- [22] Real-time FPGA implementation of Hough Transform using gradient and CORDIC algorithm, Si Mahmoud Karabernou, Faycal Terranti
- [23] Efficient Hough transform on the FPGA using DSP slices and block RAMs, Xin Zhou, Norihiro Tomagou, Yasuaki Ito, and Koji Nakano
- [24] [http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL\\_COPIES/MARBLE/low/edges/canny.htm](http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MARBLE/low/edges/canny.htm)
- [25] An Improved Canny Edge Detector and its Realization on FPGA, Wenhao He and Kui Yuan

- [26] Embedded Systems Computer Architecture (extended abstract) Jakob Engblom
- [27] Study and Comparison of Various Image Edge Detection Techniques, Raman Maini, Dr. Himanshu Aggarwal
- [28] Field Programmable Gate Array technologies for robotics application, P.H.W. Leong and K.H. Tsoi.
- [29] SLAM with Corner Features Based on a Relative Map, Manuel Altermatt, Agostino Martinelli, Nicola Tomatis and Ronald Siegwart
- [30] Improving Localization Robustness in monocular SLAM using a High-Speed Camera, Peter Gemeiner, Andrew J. Davison, Markus Vincze.