Obstacle Detection for Indoor Navigation of Mobile Robots

Master Thesis

Submitted in Fulfillment of the Requirements for the Academic Degree M.Sc.

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Md Rashedul Islam Rasel
Student ID: 362520
Date: 25.01.2017

Supervising tutor: Prof. Dr. W. Hardt
Dipl. Inf. René Schmidt
Abstract

Obstacle detection is one of the major focus area on image processing. For mobile robots, obstacle detection and collision avoidance is a notorious problem and is still a part of the modern research.

There are already a lot of research have been done so far for obstacle detection and collision avoidance. This thesis evaluates the existing various well-known methods and sensors for collision free navigation of the mobile robot. For moving obstacle detection purpose the frame difference approach is adopted. Robotino® is used as the mobile robot platform and additionally Microsoft Kinect is used as 3D sensor. For getting information from the environment about obstacle, the 9-built-in distance sensor of Robotino® and 3D depth image data from the Kinect is used. The combination is done to get the maximum advantages for obstacles information. The detection of moving object in front of the sensor is a major interest of this work.

Keywords: Obstacle avoidance, Robotino, ROS, Kinect, Navigation.
Content

Abstract .......................................................................................................................... 1

Content ......................................................................................................................... 2

List of Figures .................................................................................................................. 4

List of Tables .................................................................................................................... 6

List of Abbreviations ....................................................................................................... 7

1 Introduction .................................................................................................................. 8
  1.1 Motivation ................................................................................................................. 8
  1.2 Outline ...................................................................................................................... 9

2 State of the Art ............................................................................................................ 10
  2.1 Autonomous Navigation of Robot ........................................................................... 10
  2.2 Obstacle detection Algorithm .................................................................................. 15
    2.2.1 Infrared sensor information based algorithm ..................................................... 16
    2.2.2 Depth information and Image based algorithm ................................................ 16

3 Tools, Platform and Sensors used ............................................................................... 25
  3.1 Robot Operating System – ROS .............................................................................. 25
    3.1.1 Peer to Peer ....................................................................................................... 25
    3.1.2 Multi Language Support ................................................................................... 26
    3.1.3 Free and Open Source ...................................................................................... 26
    3.1.4 ROS Architecture ............................................................................................. 27
    3.1.5 ROS Computation Graph Level ......................................................................... 27
  3.2 Robotino® ................................................................................................................ 30
    3.2.1 Operating System .............................................................................................. 32
    3.2.2 Control ............................................................................................................... 32
    3.2.3 Sensors .............................................................................................................. 32
    3.2.4 Drive Unit .......................................................................................................... 36
  3.3 Kinect Sensor ............................................................................................................ 37
    3.3.1 Functional mechanism Kinect depth sensor ..................................................... 39
    3.3.2 Software and driver ......................................................................................... 40
3.4  Point Cloud Library .......................................................... 41
3.5  OpenCV ................................................................. 43
   3.5.1  cv_bridge .......................................................... 44
   3.5.2  image_geometry .................................................. 44
4   Implementation ........................................................... 45
   4.1  Process flow design .................................................. 45
   4.2  Connecting to Robotino® ........................................... 46
   4.3  Mapping and Navigation ............................................ 47
   4.4  Obstacle Detection .................................................. 51
      4.4.1  Obstacle detection by Distance Sensors .................. 53
      4.4.2  Obstacle Detection by Kinect .............................. 54
      4.4.3  Detect Moving Obstacle ................................... 55
      4.4.4  Combined decision for obstacle detection ............. 62
5   Evaluation ............................................................... 64
   5.1  Implementation Result ............................................. 64
   5.2  Further Scope ...................................................... 68
6   Conclusion ............................................................... 70
7   Appendix ................................................................. 71
   7.1  Obstacle detection by IR Sensors ................................ 71
   7.2  Obstacle detection by Kinect Sensors .......................... 72
   7.3  Moving Obstacle detection by Kinect Sensors .............. 75
Bibliography ......................................................................... 81
List of Figures

Figure 1 : Dijkstra’s pseudo code [64]. ................................................................. 11
Figure 2 : A* pseudo code [36]. ........................................................................... 12
Figure 3 : DWA Local Planner pseudo code [2]. ...................................................... 13
Figure 4 : Hypotenuse of Right angle triangle. ....................................................... 14
Figure 5 : Obstacle avoidance algorithm [21]. ........................................................ 16
Figure 6 : Background Subtraction Algorithm for Moving Object Detection [35]..... 18
Figure 7 : Flow chart for moving object detection using BBME [43]. .................... 19
Figure 8 : Contour Extraction Algorithm [45]....................................................... 20
Figure 9 : Descriptor Algorithm [45]. .................................................................. 21
Figure 10 : Merging of technique [45]. ................................................................. 22
Figure 11 : The generic RANSAC algorithm [47]. ................................................ 23
Figure 12 : Multi language communication. ......................................................... 26
Figure 13 : ROS Architecture. .............................................................................. 27
Figure 14 : ROS Computation Graph Level........................................................... 28
Figure 15 : ROS Topic .......................................................................................... 29
Figure 16 : ROBOTINO®. ....................................................................................... 30
Figure 17 : Embedded PC ROBOTINO® [48]. ....................................................... 32
Figure 18 : ROBOTINO® Bumper Sensor [48]. ..................................................... 33
Figure 19 : ROBOTINO® 9 Distance Sensors arrangement. .................................... 34
Figure 20 : ROBOTINO® Distance Sensor characteristic [48]. ......................... 35
Figure 21 : ROBOTINO® Camera Sensor [48]. ...................................................... 36
Figure 22 : Drive unit ROBOTINO® [48]. ............................................................. 36
Figure 23 : Drive unit of Robotino® ..................................................................... 37
Figure 24 : Kinect XBOX 360 Sensor. ................................................................. 38
Figure 25 : projecting a pattern of dots in infra-red from RGBD Sensor ............... 39
Figure 26 : Depth Map Image .............................................................................. 40
Figure 27 : PCL modularity graph [49].................................................................. 42
Figure 28 : cv_bridge between ROS and OpenCV............................................... 44
Figure 29 : ProcessArchitecture. ......................................................................... 46
Figure 30 : ROS launch command ....................................................................... 47
Figure 31 : Depth image to laserscan conversion process. [51]............................. 48
Figure 32 : Mapping process. ............................................................................... 49
Figure 33 : Mapping commands and steps. ........................................................... 50
Figure 34 : Navigation simulation using rviz. ....................................................... 51
Figure 35 : Efficiently obstacle detection with multiple sensors. ....................... 52
Figure 36 : Obstacle detection process. ............................................................... 53
Figure 37: Obstacle detection process. .................................................................54
Figure 38: Real image from web cam (Left) and Depth image from Kinect (Right). .55
Figure 39: Disparity image-color mapped depth image. ..................................56
Figure 40: Moving Object Detection algorithm .............................................57
Figure 41: OpenCV Gaussian Blur sample input and output. .........................58
Figure 42: Frame difference of non-moving objects with static position of camera. .58
Figure 43: Frame difference of moving objects with static position of camera. ....59
Figure 44: Frame difference image (Left), Threshold Image (Right) .............60
Figure 45: Threshold binary image (Left), Output of morphological analysis (Right). .................................................................60
Figure 46: From Morphological dilation image (Left) to draw contour image (Right). 61
Figure 47: Combined Decision. .........................................................................62
Figure 48: Convex Hull .........................................................................................63
Figure 49: Obstacle detection observation .......................................................64
Figure 50: Kinect Movement Detection Result with static Obstacle ..................67
Figure 51: Kinect Movement Detection Result with moving Obstacle ............68
List of Tables

Table 1 : Robotino®Configuration ........................................................................................................31
Table 2 : Kinect Specification [69][70]. ..............................................................................................38
Table 3 : List of OpenNI nodes.............................................................................................................41
Table 4 : PointCloud and PointCloud2 message type definition [71]. ..............................................43
Table 5 : PointCloud2 and LaserScan message type definition [71]. ..............................................48
Table 6 : Obstacle message type Definition. .........................................................................................62
Table 7 : Test objects and results for IR sensor ......................................................................................65
Table 8 : Test Objects and results for Kinect Sensors...........................................................................65
Table 9 : Test results for Movement detection by Kinect. .................................................................66
Table 10 : Test Observation Success Rate for IR Sensors .................................................................66
Table 11 : Test Observation Success Rate for Kinect .........................................................................67
Table 12 : Test Observation Success Rate for Moving Obstacle detection ......................................67
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGV</td>
<td>Autonomous Ground Vehicle</td>
</tr>
<tr>
<td>BBME</td>
<td>Block-Based Motion Estimation</td>
</tr>
<tr>
<td>CD</td>
<td>Cell decomposition</td>
</tr>
<tr>
<td>DWA</td>
<td>Dynamic Window Approach</td>
</tr>
<tr>
<td>ICA</td>
<td>Independent Component Analysis</td>
</tr>
<tr>
<td>MCL</td>
<td>Monte Carlo Localization</td>
</tr>
<tr>
<td>OpenCV</td>
<td>Open Source Computer Vision Library</td>
</tr>
<tr>
<td>PRM</td>
<td>Probabilistic Road Map</td>
</tr>
<tr>
<td>ROS</td>
<td>Robot Operating System</td>
</tr>
<tr>
<td>SLAM</td>
<td>Simultaneous Localization and Mapping</td>
</tr>
<tr>
<td>UKF</td>
<td>Unscented Kalman Filter</td>
</tr>
<tr>
<td>ANFIS</td>
<td>adaptive neuro-fuzzy inference system</td>
</tr>
<tr>
<td>BSD</td>
<td>Berkeley Software Distribution</td>
</tr>
<tr>
<td>CPP</td>
<td>Classical path planning</td>
</tr>
<tr>
<td>GPP</td>
<td>Global Path Planner</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>MOG</td>
<td>Mixture of Gaussians</td>
</tr>
<tr>
<td>PCA</td>
<td>Principal Component Analysis</td>
</tr>
<tr>
<td>RANSAC</td>
<td>Random Sample Consensus</td>
</tr>
<tr>
<td>SAD</td>
<td>Sum of Absolute Difference</td>
</tr>
<tr>
<td>TEB</td>
<td>Timed Elastic Band</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Motivation

Robotics is referred to the scientific research and study of robots and their technology, design, manufacture, and applications [58]. A large number of areas need further research related to robotics, for example: robot mapping, scalable architectures, navigation, planning, and world modeling, etc. Navigation is one of the critical ability for robot to realize and navigate surrounding environments. The robot is claimed to be mobile if the robot can navigate free from collision with obstacles. Mobile robot can react based on the knowledge and information gathered by the sensors from the real environment and based on that it can reach to the goal position efficiently and reliably. Navigation process involves sensing data from the environment, acting based on the sensor data, planning strategy, architecture, hardware, computational and power efficiencies, etc.

For collision free robot navigation, obstacle detection and avoiding the obstacle is a very important. Recent research has attempted to improve upon obstacle detection and obstacle avoidance methods by investigating various types of sensors like camera, sonar sensors, laser scanners, Infrared sensors, etc. There are some limiting factors for deployment of sensors on small inexpensive robotic platforms considering efficiency, power and cost of those sensors. In the year 2010 Microsoft Corporation released XBOX Kinect, which is a low cost, easily affordable three-dimensional (3-D) depth ranging sensor. Kinect is nearly one tenth of a price of laser range finder, which provides a cost-effective solution using infrared light to generate voxel (depth pixels) and consequently adding a third dimension to the image received by the sensor. Several researches have shown that, Xbox Kinect is to be an invaluable device for modeling of indoor environments for robot navigation. Also the Kinect has been shown to have significant success in separating an object from background or surroundings [56]. Although the Kinect has some limitation including the depth sensing range, projection pattern on very bright and shiny surface, it is still one of the best existing solution for object detection and collision avoidance in indoor environment.

This thesis contributes to the area of mobile robotics, navigation and obstacle detection and the purpose is to provide a proof-of-concept utilizing existing resources to develop and implement an obstacle detection and avoidance system on the mobile robot platform Robotino®.
1.2 Outline

This thesis is organized into seven different chapters. Each chapter contains one or more sub-sections. A background study of the pertinent research on the indoor mobile robot navigation, obstacle detection using Xbox Kinect, moving obstacle detection and its uses on mobile robotics, state-of-the-art algorithms are provided in Chapter II.

The software, hardware, tools and other platforms that are used are discussed on chapter III. Chapter IV describes about the implementation process of the system. After implementation of the proposed system, the output result of the system is evaluated and discussed in chapter V. The future scope and conclusion is discussed in Chapter VI. Code snippet are added as an appendix in chapter VII.
2 State of the Art

Obstacle detection is a salient feature of robot navigation. Obstacle detection and efficiently avoiding the obstacle is very much important for smooth autonomous navigation of a mobile robot. Although there are so many researches have been done about collision free robot’s autonomous navigation but still it's a challenge for smoothly avoiding obstacles in real world.

Navigation for autonomous robot is done in several ways, for example known environment navigation with map and another is for unknown environment. But in both cases in real world there are always some objects which are not known for the robot and considered as obstacle. During autonomous navigation both in known and unknown environment, detecting potential obstacles and efficiently avoiding those obstacles is a big challenge. Obstacles could be both stationary and moving objects. The navigation algorithm plans path and navigate but also it sense information from the environment to detect potential dynamic obstacles and also the robots position estimation is sensed from the sensor. The approach and algorithm for navigation and obstacle avoidance are discussed below section.

2.1 Autonomous Navigation of Robot

From technical point of view, robot navigation focuses on primarily generating optimal global paths to move the robot from source to destination in a real time environment. In this section different algorithms and approaches for robot navigation are briefly introduced. For navigation, the robot needs to plan global path which is from source to destination in the map (or world). Another one is local path, which is a small frame or cell around the robot to detect any obstacle, so that the robot can reactively correct the global planned path. The process for autonomous robot navigation to senses the information from the environment and plans accordingly a collision-free trajectory to navigate to a destination is called Global path planning (GPP) [37][63].

The robot creates a two-dimensional geometric representation of its environment using the data provided from the sensor (i.e.: laser, Kinect, ultrasound etc). The robot also needs to determine the current location on real world, for that purpose odometry data is used which is supplied from the robot's wheel encoders. For GPP, the Dijkstra’s algorithm [64] is given below in Figure1.
Aleksandar Tomović mentioned about a modified A* algorithm for path planning [36]. The map navigation use A* algorithm (shown in figure 2) which combined Dijkstra’s algorithm and Best First search algorithm to find the shortest path. Besides that, there are several other concepts related to robot navigation and motion planning. Among them the “Following vehicle” concept was introduced by Classical path planning (CPP). CPP represents the world called the configuration space [38] [39], which uses generalized coordinates. Cell decomposition (CD) methods are very popular in robotics application for transforming the configuration space into cell regions and robot motion planning [37].

![Figure 1: Dijkstra's pseudo code [64].](image-url)
Localization comes after Global path planning, which means the robot must know its own position, orientation or moving direction on the map. The robot must update its local position continuously. For local trajectory planning, there are mainly three common approaches.

- Potential Field based: It is an attraction-repulsion based approach where goal or target point has attraction and obstacle has repulsion force.
- Dynamics based: This approach consider the dynamic or velocity of the robot to find a solution for collision avoidance and path planning.
- Sampling based: In this approach, the unoccupied states from the map is referenced and combined for a solution.

![Figure 2: A* pseudo code [36].](image)
Dieter Foxey, Wolfram Burgardy, Sebastian Thruny describes et al [2] an algorithm named Dynamic Window Approach (DWA) to reactive collision avoidance for mobile robots. DWA is directly derived from the dynamics of the robot. Figure 3 shows the pseudo code of the DWA algorithm.

```
1 BEGIN DWA (robotPose, robotGoal, robotModel)
2   desiredV = calculateV(robotPose, robotGoal)
3   laserscan = readScanner()
4   allowable_v = generateWindow(robotV, robotModel)
5   allowable_w = generateWindow(robotW, robotModel)
6   for each v in allowable_v
7       for each w in allowable_w
8           dist = find_dist(v, w, laserscan, robotModel)
9           breakDist = calculateBreakingDistance(v)
10          if (dist > breakDist) //can stop in time
11             heading = hDiff(robotPose, goalPose, v, w)
12             clearance = (dist-breakDist)/(dmax - breakDist)
13             cost = costFunction(heading, clearance, abs(desired_v - v))
14         if (cost > optimal)
15             best_v = v
16             best_w = w
17             optimal = cost
18       set robot trajectory to best_v, best_w
19 END
```

Figure 3 : DWA Local Planner pseudo code [2].

Monte Carlo localization (MCL) is a well established and very popular algorithm for robot localization using a particle filter. The algorithm is able to estimates the current position and also the orientation of a mobile robot in real world with reference to a given map while it moves and senses from the environment [41]. MCL is also known as Particle filter localization.

Simultaneous Localization and Mapping (SLAM) is one of the most popular methods for localization approach which was during an automation conference by introduced by IEEE Robotics [40]. Using SLAM, the robot learns continuously from environment using various sensors to locate its correct position [65]. The robot can reach to its destination efficiently and quickly as the robot learns localization. During creating map using robot and its sensor, SLAM approach is used. SLAM Estimates the distribution over robot poses.

There are several other methods available for motion planning like Sampling Based Planning, Grid Based Planning, and Reward-Based Planning. Sean Quinlan et al [3] mentioned about the modification of collision-free paths, which introduced a new
framework to reduce the gap between GPP and sensor-based robot control [66]. A very modern and effective approach was introduced called Elastic-Band, which is flexible string-like object and was introduced to determine the modification of a path. Rosmann et al. [4] and [5] presents physically-based further extension to the elastic band called Timed Elastic Band (TEB). It optimizes the global trajectory on runtime, minimizes the trajectory execution time and avoiding obstacles. The optimization based online planner locally refines the initial coarse path and continuously deforms the trajectory. With the changing position of obstacles especially with the moving obstacle, this trajectory deform process fails to calculate the local trajectory [1]. Also the robot's physical movement does not always give position coordinate with 100% accuracy that means the robots position has a very little amount of tolerance. Therefore it is very difficult to find the exact goal position and also it can result in the robot moving around its destination point to find the exact location [67]. If two moving obstacles approaching from opposite direction then the robot is forced to stop which is related to the freezing robot problem occurs [36].

The path planning with uncertainty or for unknown environments is bit different. As there is no map or world, it is not necessary to create SLAM based map or GPP. It only needs localization and destination co-ordinates. In [21], Straight Line trajectory Adaption method is proposed for robot navigation in unknown environment. When the robot moves from initial or start position towards the goal or destination point, the starting points coordinates \((x, y)\) along with orientation are set to values \((0, 0, 0)\), wherever the robot aims to resume its movement toward its goal after avoiding any obstacle. At this stage the robot will calculate the distance of the destination and also the rotating angle by adapting the straight line equation before next movement.

![Figure 4: Hypotenuse of Right angle triangle.](image-url)
\[ C = \sqrt{(XA - XB)^2 + (YA - YB)^2} \ldots \quad (1) \]

Equation 1: Hypotenuse of triangle for two known points.

\[ \alpha = \text{atan2}(X, Y) \ldots \quad (2) \]

Equation 2: Orientation angle.

Equation 1, 2 is for calculating the hypotenuse for straight line calculation and the required angle for rotating direction. When the robot starts moving and senses any obstacle within the distance, it either moves left or right based on a decision from pre-defined lookup table. After it moves to avoid collision from the obstacle it again calculates its rotating angle to drive toward the destination point and again resets its starts position from there as (0, 0, 0). It repeats the process until the destination has been reached.

2.2 Obstacle detection Algorithm

Obstacle detection is one of the most important features for collision free, smooth and efficient autonomous navigation of mobile robot. Although navigation is done by several navigation approaches (mapping, path planning) but it depends on sensors feedback to collect data from the real environment and dynamically make proper decision for obstacle detection and avoidance. There are different types of sensors available and used for obstacle detection depending on the system design and usability. Distance sensor, ultrasound sensor, camera (webcam, rgb camera), laser sensor, depth camera sensor, etc. are widely available for detecting objects within different distance range. Every type of sensors is specialized with its own capability and can perform with an acceptable level of accuracy in real life implementation. But combining multiple types of sensors for obstacle can come up with better dimension of accuracy and usability, which is refer as sensor fusion. And previous researches shows that sensor fusion has improved decision making process of routing the mobile robot and detection of obstacles [20]. A hybrid mechanism was introduced by Hui, V. Mahendar where for collision avoidance neuro-fuzzy controller was used [16]. Moreover, an adaptive neuro-fuzzy inference system (ANFIS) was applied for an autonomous ground vehicle (AGV) to safely reach the target while avoiding obstacles by using four ANFIS controllers [17]. Another sensor fusion based on Unscented Kalman Filter (UKF) was used for mobile robots’ localization problems. To obtain
data from the environment for the fusion algorithm different types of sensors like accelerometers, gyroscopes were used. For this thesis implementation purpose infrared distance sensors and 3D camera sensor based depth information has been used. The below sections will describe brief description of the approaches.

### 2.2.1 Infrared sensor information based algorithm

Infrared (IR) Distance Sensors are small sized, easy to use and gives different measurement ranges of objects information from the environment. Dr. M. Ali and Tariq Ali et al [21] presents a simple approach to detect obstacle using distance sensors within the range of 4-40 cm distance. The robot continuously sense data using IR sensor from the environment and the data is converted into binary logic based on distance measurement. After that a lookup table is used to set the movement command of the robot. The lookup table consists of simple information which is logically converted for moving direction and the robot speed. For the sensors, there is a dependency graph for voltage value to distance of the reflective object which is given below as figure 5.

If output voltage from infrared sensor ≥ 0.9 V //scale conversion of (0.9v = 14 cm)

Then analog voltage value is set to (1)

Else set it to (0)

**Figure 5 : Obstacle avoidance algorithm [21].**

In this work we will use distance sensor equipped with Robotino®, to detect obstacle within short range around 40cm.

### 2.2.2 Depth information and Image based algorithm

Vision based sensing has always a high potential for a mobile robot to obtain static or moving obstacle information from its surroundings. Computer vision is an active field of research and a huge number of researches have been done for object detection. For smooth autonomous navigation of a robot, object detection and efficiently avoid the obstacle is very much important. Traditional camera (i.e.: web cam) can provide us with 2d image. It is very difficult to estimate the object distance from the camera and also from color image calculating the object and tracking the movement is also very much expensive in context with the calculation and processing time. 3D camera based depth sensors provide with image and depth information of the frame from
where distance of each pixel can be acquired and thus obtain the object distance information. With the 3D depth image in grayscale format it is also possible to extract a lot more image feature rather than normal rgb image, for example identifying if any obstacle is moving or stationary and even more to find the moving direction of the object.

Background subtraction is a well-known approach for computer vision based object detection. Therefore, there are already a lot of works in the literature focused on it. Simple models for static backgrounds [22–24], more advanced methods such as MOG (Mixture of Gaussians) [25–27], Bayesian decision rules [28], the Codebook-based model [29, 30], Kernel density estimation [31] or Component Analysis like Principal Component Analysis (PCA), Independent Component Analysis (ICA) [32, 33] are capable of dealing with dynamic backgrounds.

Enrique J. Fernandez-Sanchez and Javier Diaz et al [34] describes about how to improve the object and movement detection by fusing depth and color input with the classic background subtraction method. One of the most important features is to detect moving obstacle. Kuihe Yang, Zhiming Cai, Lingling Zhao introduces et al [35] different approaches like background subtractions based on codebook model and Bayesian Classification, frame difference (two frame, three frame), etc. To detect any kind of change from a camera vision or video, the difference of two consecutive frames is one of the most effective approaches. The frame difference can be defined as equation 3:

\[
D(x, y, t + 1) = \begin{cases} 
1 & \text{if } f(x, y, t) - f(x, y, t + 1) > th \\
0 & \text{otherwise}
\end{cases} \quad \cdots \cdots (3)
\]

Equation 3: Frame Difference principle.

\(f(x, y, t)\) is a frame at time \(t\). The immediate next frame at \((t+1)\) is \(f(x, y, t+1)\) and \(th\) represents the threshold value parameter for decision. Three Frame difference is an improved method from two frame difference and it uses three consecutive frames to find the difference. From the difference of three frame images, the changes or movement detection is done using the contour object. Equation 4, 5 are for three frame difference method. The additional option is that there is a scope for adding multiple threshold value. For \(n\) frame difference method there is an option to use \((n-1)\) threshold values which can give more flexibility to find the moving contours.
Equation 4: Difference of first two frames.

\[
b_{k-1,k} = \begin{cases} 
0 & \text{background } |f_k(x,y) - f_{k-1}(x,y)| > T_1 \\
1 & \text{foreground } |f_k(x,y) - f_{k-1}(x,y)| \leq T_1 
\end{cases} \quad \ldots \quad (4)
\]

Equation 5: Difference of second and third frame.

\[
b_{k,k+1} = \begin{cases} 
0 & \text{background } |f_{k+1}(x,y) - f_k(x,y)| > T_2 \\
1 & \text{foreground } |f_{k+1}(x,y) - f_k(x,y)| \leq T_2 
\end{cases} \quad \ldots \quad (5)
\]

\(f_k(x,y)\) is the k-th frame. \(T_1, T_2\) is thresholds with different values. \(b_{k-1,k}\) and \(b_{k,k+1}\) are the binarization of two adjacent frames difference.

Other than frame difference, Background Subtraction is very popular and simple method to identify a moving obstacle. The algorithm proposed in [35] is as below mentioned in figure 6:

\begin{itemize}
    \item \textbf{Step one:} Calculate the absolute gray image of the current frame image and the background image.
    \item \textbf{Step two:} By setting the threshold \(Th\), obtain the binarization image, thereby extracting the moving object region from the image.
    \item \textbf{Step three:} Apply mathematical morphology to filter the difference image, and then analysis communicating region. If the area of connected region is larger than a given threshold value, then that is a moving object.
\end{itemize}

\textbf{Figure 6 : Background Subtraction Algorithm for Moving Object Detection [35]}

The background subtraction process is very much effective when the camera is in a fixed position and a fixed background can be defined. For example, security surveillance cameras are fixed and an image is set as a background of that place. Block-Based Motion Estimation (BBME) was presented by Jeongdae Kim, Yongtae Do et al [43] with some preliminary result of detecting moving obstacles by using camera attached on indoor mobile robot. As camera is attached to mobile robot, classical approaches introduced above will not be effective as movement of camera will always give images from different angel and position. Thus, images captured from the moving camera cannot be compared with a fixed background (Background subtraction) or consecutive frame (frame difference). The BBME estimation divides the images into several small blocks and motion is compared for each small block for consecutive two frame images. BBME has been widely used particularly in video coding. This approach can be implemented optimally for real time processing for two
different images. The BBME method firstly divide image frame into non-overlapping blocks. Then motion vector for the blocks are determined to minimize the value of matching criteria. To obtain this, Sum of Absolute Difference (SAD) method [59] [60] is used, which calculates between the current and previous image block.

\[
SAD = \sum_{x=1}^{s} \sum_{y=1}^{s} |B_t(x, y) - B_{t-1}(x, y)| \quad \ldots \quad (6)
\]

Equation 6: Sum of Absolute Difference [43].

The motion vector is represented by the spatial difference, which can be regarded as disparity if the motion is horizontal. Then a vertical projection is done over the disparity image and moving object is detected from the current frame. Figure 7 represents the flow chart diagram of detecting moving object using BBME method.

![Flow chart for moving object detection using BBME](image)

Figure 7 : Flow chart for moving object detection using BBME [43].

In the block matching part of the above summarized algorithm, a function from OpenCV named cvFindStereoCorrespondenceBMIs used. The method can find best matching blocks in the images of two different stationary cameras located in different position. But here the function is used for two different frames from the same camera. However, the search in both cases is to serve same purpose which actually finds similarity of corresponding blocks in the two slightly different images. Using this approach, it can detect moving object but if the object is far distance then it may not detect properly the motion. Also, color and light reflection affect the performance of the algorithm.
To detect moving objects from optical flow-field an algorithm was described in [44], which compares measured optical flow vectors with the generated vectors. It iteratively generates optical flow vectors for different possible real world coordinates of the objects in the scene. The proposed algorithm was developed as a part of multi-sensor fusion system, so that the computation is not so demanding due to CPU time sharing with several sensors.

```
openCV formatimage = convertOpenCV (image received)
image = convert2OpenCV (BLACKWHITECODE, image)
image = openCVEqualize(image)
MatFormat image 2 = convertMatFormat(image)
Erode(image 2)
Dilate(image 2)
for pixel i do //This if for all pixels in the image
    if (isTooFar(pixel i)) then
        pixel i ← 0
    else
        pixel i ← 1
    end if
end for
contour selected ← getBiggerContour(arrayContours)
getBoundingBox(contour selected)
correlate(image 2, models, results)
Point extrema = getExtrema(results)
if ((extrema.x > extrema.y)OR(extrema.x > extrema.z)OR(rangeFullFills())) then
    print("ObjectDetected!")
end if
```

Figure 8: Contour Extraction Algorithm [45].

Contour Extraction Concept, Descriptor extraction concept and combination of both of them to detect object in a depth sensor based system was nicely described and evaluation of the result on a ROS based implementation in a real-life robot was presented in [45]. The implementation system was consisting of a robot called Turtlebot, a RGBD camera Kinect, OpenCV and Robot Operating System. The algorithms are implemented with the use of OpenCV libraries. The contour extraction implementation algorithm pseudo code [45] is given in figure 8. And the Descriptor Algorithm pseudo code [45] is given in figure 9.
Both Contours and Descriptors are having advantages and disadvantages. Contours are useful to find a shape clearly represents an object. Also, it is not computationally heavy and allowing a fast response. The added advantage of using depth image from the Kinect is that the light changes do not affect the method used. The problem with Contour is it cannot discriminate a shape of any object. On the other hand, Descriptors provide the ability to detect specific objects. But the problem is that it works for specific objects, not generalized. Therefore, a combination of both the technique was proposed. There are several techniques for combination.

- **Switching**: The ability to change from one method to another based-on condition. Whenever any object is detected it must pass through a condition to switch either contour or descriptor.
- **Merger**: This is a combined output from the both algorithm. The result is a merged from both contour and descriptor.
- **Accuracy**: This implies the ability define which one is better for that scenario. It takes both output and decides the better one.

For combining technique, among switching, merging and accuracy any one of the technique must be selected. Based on the system implementation they choose merging technology as show in figure10 below.
Using depth sensor based data moving object detection for mobile robot based on frame difference concept is proposed on [46]. As mentioned earlier on this section frames difference is a common approach for detecting the changes of two consecutive frame data. To directly compare to successive, image the frames coordinates system must be common, which is defined as registration. But when the camera is moving itself, every frame captured will not have the same registration that means the frames are in different coordinates. Also, directly comparing images by frame difference will give some error as images will contain a high amount of noise information. To overcome these issues three step processing is done. Registration, Frame Differencing and segmentation are the three steps. In the registration step noise filtering and removing outlier is done. Random Sample Consensus (RANSAC) method is used for removing the outliers or the false matching. Figure 11 shows the generic RANSAC algorithm [47].
Figure 11: The generic RANSAC algorithm [47].

For the frame difference step, subtraction is performed using the voxel. Voxel is defined as a value of regular grid in a three-dimensional space [61]. As the vision system is attached on the body of the indoor mobile robot, the camera in this case is not stationary. And for detecting moving obstacle with moving camera with the robot, there will not be any fixed background image. It is not wise or useful to implement background subtraction or frame difference directly for a system where the camera is
not in a stationary position. For this thesis, the approach will be continuous frame difference, where the sensor will continuously take image frame and compare the differences between the images.
3 Tools, Platform and Sensors used

For implementation purpose, we have used robot, sensors, software and other necessary tools. Below sections separately introduces and briefly describes about the used robot, tools, software, etc.

3.1 Robot Operating System – ROS

It is clearly understandable that to implement an indoor navigation system in a mobile robot platform, the developer must use robot, different type of sensors, devices and also will implement some algorithm, logical program. But the first issue is to put all the things together so that the system works and therefore a platform or framework is required. Robot Operating System (ROS) is an open-source, heterogeneous and scalable P2P network-based robotics framework. ROS provides wide range of libraries and tools to help software developers creating innovative robot applications in an organized pattern. The benefit of using ROS is to get some useful essential features which will reduce the development time and complexity by providing device drivers, libraries, visualizers, message-passing, package management, access to third party tools, hardware abstraction, and more. ROS is licensed under an open source, BSD license. The basic advantage of ROS is code reuse for the robotic developer and researcher in a common and generalized platform. ROS is a language-independent architecture (C++, python, lisp, java, and more), scalable platform (ARM CPUs to Xeon Clusters) and intent to enable researchers to rapidly develop new robotic systems by increasing the reusability through use of standard tools and interfaces.

There are not many Robot frameworks out there which support the development of general purpose robot software across many platforms. This makes it difficult to accomplish even simple and trivial tasks which might seem simple to humans but the Robot developer should navigate and sort through many variations between instances of tasks and environments. Having a general-purpose framework like ROS helps us to create complex and robust robot behavior across many platforms which have got the support of ROS. Below section will point out some key feature of ROS.

3.1.1 Peer to Peer

ROS consists of small programs which are connected to each other and they also exchange messages. These messages are sent within the ROS framework and the travel directly from one program to another. They are not routed through any central
routing service. This will be an advantage to scale up the system when the amount of data increases.

3.1.2 Multi Language Support

ROS supports a number of programming languages such as C, C++, Java, Python, etc. ROS client libraries exist for C++, Python, Java, JavaScript, Ruby, OpenCV and many more. Scripting languages such as Python and Ruby makes it easier to accomplish many software tasks easily compared to others. A simple graphical representation of multi language communication is shown by the figure12 below.

![Multiple Languages](image)

3.1.3 Free and Open Source

ROS is released and licensed by BSD (Berkeley Software Distribution) which can be used for commercial and non-commercial purposes. ROS has IPC that is inter-process communication process which passes the data between modules. So, various components can have individual licenses. People who use ROS for commercial purposes can do their development behind a firewall and still have their closed source modules communicating with large number of open source modules.
3.1.4 ROS Architecture

Once the Robot operating system is installed on Linux, it is important to know the architecture design. ROS architecture is designed and divided into three sections and levels of concepts. Figure 13 is the diagram of ROS architecture and below the main levels is briefly described.

- The File system level: This is the first level. There are group of concepts in this level which is used to explain ROS folder structure, how ROS internally works, and the minimal files needed for it to work.
- The Computation Graph level: In the second level communication between processes and systems happen. If there is more than one computer ROS sets up systems which can handle all the processes which can communicate with computers on the network.
- The Community level: This is external to the ROS in the sense that is has nothing to do with operation of ROS. However this level is needed to explain tools and concepts and where any developer can share his/her knowledge, algorithms developed or code written for ROS. This level is important in the context of helping ROS to grow quickly.

![Figure 13: ROS Architecture.](image)

For understanding more about ROS, the sub-section below will briefly go through the Computation graph level.

3.1.5 ROS Computation Graph Level

The second level is a network where all the processes are connected. Figure 14 shows a graphical concept view of Computation Graph Level or ROS Architecture.
Any node which is created can access the network, interact with other nodes and see the information which is being sent and transmitted to the network. All the concepts mentioned in the figure 14 nodes, Master, Parameter server, Messages, Topics, Services, and Bags provide data to the graph in different ways. A brief for each one of them is discussed in the below section.

I. Master
A master node is requirement to run all other nodes present in the system. Master node provides name registration. Communication is not possible with other nodes, services, messages without master node. One master node on one computer can serve all other node running on different computers when they are connected. ROS Master Node acts as a name service in the ROS Computation Graph. The Master Node stores all the active topics and services registration information.

II. Node
This is the basic concept in the second level. This is where computation is done. If there is a need to communicate with other nodes in the network then separate node has to be created with the process to be connected to the ROS network. A robot can have many nodes which provide different functions to control it. A node containing all functionalities of the robot can be written but it is usually preferred to write a node which can provide a single functionality. Nodes are written with an ROS client library or with catkin workspace. When a node is executed, it communicates with the Master node for registration the information. Once the node is registered with the master node, it can communicate with all other nodes registered with the same master.

Figure 14: ROS Computation Graph Level.
III. Parameter Server
A central location is used to store data in ROS. And Parameter server helps us to store that data using keys. Nodes can configure while it is running with Parameter server.

IV. Messages
An interface is required for the Nodes to communicate and Messages provide that facility. A message contains data which can be sent to other nodes. The custom or user defined message should be described in the message folder so that any other node can get the access of the definition of message type.

V. Topics
Every message contains a name using which it is routed by the ROS network. If there is a data sent or received by a node, then it is usually said a node is publishing or subscribing a topic respectively. A node can subscribe to a topic and there is no necessity that a topic should exist for this node which can be publishing data. Each topic should have a unique name to avoid conflicts. Figure 15 shows a graphical representation of relation between node and topic.

![Figure 15: ROS Topic.](image)

For example, in the above figure, *teleop_turtle* and *turtlesim* is two different nodes. The node *teleop_turtle* is publishing move command via keyboard for the robot. On the other side the node *turtlesim* is simulating the movement of the robot in graphical interface and it receives the move command. The first node is publishing command information to a topic called */turtle1/command_velocity* and the second node is subscribing the command information from the same node. If both the node are registered with the same master then only it is possible to subscribe and publish with the same topic. There could be more than one publisher nodes are publishing data into same topic and on the other hand there could be multiple subscriber nodes that are subscribing data from the same topic.
VI. Services
A node consists of a service; all other existing nodes can communicate with it by the help of ROS client libraries.

VII. Bags
Bags are a method of recording and saving data which can be played back. Any kind of data can store such as sensor data and played back when necessary.

3.2 Robotino®

The mobile robot system Robotino® is a robot platform mainly used in education and research purpose. This platform is with an open mechanical interface for the integration of additional mechanical device and equipped with a drive system allowing for Omni-directional moves. It has distance, infrared and inductive sensors, and also features a camera with VGA resolution. Figure 16 shows the image of Robotino® taken from the Festo-didactic website.

![Figure 16: ROBOTINO®](image-url)
Robotino® comes with real-time Linux operating system. It can be controlled remotely via wireless LAN. However, the limited computational capacity of Robotino® prevents the implementation of extensive computational tasks like complex image processing, machine learning based applications, etc. That is why Robotino® is mostly operated remotely. An advantage of using Robotino® is its flexibility: users can easily explore the robot's functionality and intuitively understand how changes to the program are reflected in the robot’s behavior. The basic hardware components of the robot are fixed, that is why the remote application does not need to be adapted when modifying the program. The software Robotino® View is a special version of visualization software that only supports remote execution with a fixed hardware model. The configuration information of Robotino® is given in a table below.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel i5, 2.4GHz, dual-core</th>
<th>Diameter</th>
<th>450mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>8GB RAM</td>
<td>Weight</td>
<td>20KG(approx.)</td>
</tr>
<tr>
<td>WLAN</td>
<td>Yes</td>
<td>Drive</td>
<td>Omni Directional</td>
</tr>
<tr>
<td>Control</td>
<td>32 bit microcontroller and free motor connection</td>
<td>Bumper</td>
<td>Rubber Protection with anti-collision sensor</td>
</tr>
<tr>
<td>Ethernet</td>
<td>2xEthernet</td>
<td>IR Sensor</td>
<td>9x IR Distance Sensor</td>
</tr>
<tr>
<td>USB</td>
<td>6xUSB 2.0</td>
<td>Camera</td>
<td>Full HD 1080p with USB Interface.</td>
</tr>
<tr>
<td>PCI Slot</td>
<td>2xPCI Express</td>
<td>API for Programming</td>
<td>C / C ++, JAVA, .Net, LabVIEW, MATLAB \ Simulink, ROS SmartSoft and Microsoft Robotics Developer Studio</td>
</tr>
<tr>
<td>VGA</td>
<td>1x VGA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I/O</td>
<td>1x I/O Interface</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Robotino® Configuration.

The below section will describe briefly about the hardware, software and sensors equipped with Robotino® and their functionality.
3.2.1 Operating System

The Robotino® is delivered with Linux operating system - Ubuntu installed. To work with that ROS, ROBOTINO-STACK other tools need to be installed and by connecting to the internet it can be done with normal Linux based installation command.

3.2.2 Control

The Robotino® control unit includes a controller PCB with an embedded PC and microcontroller. In the control unit the embedded PC is mounted over the main PCB and is connected to the interfaces of control unit’s and on the main PCB the microcontroller is mounted [68]. It monitors voltage supply, controlling the motor, etc. It is connected to all the motors and incremental encoders and the embedded PC. An FPGA is also used.

3.2.3 Sensors

As mentioned earlier, Robotino® is equipped with different type of modern sensors. The short description is given below.
3.2.3.1 **Bumper**
Bumper or collision safety sensor is a pressure-sensitive bumper which is placed at the bottom of the Robotino® chassis. It consists of plastic profile of varied shape and a switching chamber is integrated, which encloses two conductive surfaces. When pressure is applied on the bumper surface the surface is short circuited and a signal is generated.

![Figure 18: ROBOTINO® Bumper Sensor [48].](image)

3.2.3.2 **Distance Sensors**
Robotino® is equipped with 9 infrared sensors around its base at 40° angle to each other. Figure 19 shows the top view of IR sensors orientation fitted into the robot.
It is possible to determine distance of any object that falls in front of IR sensors in the range of 4 to 30 cm. The figure 20 below shows the relation between voltage signal value and distance to a reflective object by the distance sensor.
3.2.3.3 Gyroscope
Gyroscope is used to increase the accuracy of the position sensing and determines the change of orientation. A gyroscope sensor is mounted and connected to the PCB of Robotino®.

3.2.3.4 Camera
Robotino® is equipped with a camera screwed onto the front panel and connected to control by USB port. The camera Logitech® HD Pro Webcam C920 and the resolution, camera parameters can be configured. The camera generates live images, which can be used for navigation, obstacles and objects detection, mapping, etc.
3.2.4 Drive Unit

Robotino® drives Omni-directional. It has three independent drive units. Each drive unit consists of motors, an incremental encoder, wheels, gear unit and are integrated into the chassis of the Robotino®. Maximum speed and acceleration can be controlled programmatically and from the web interface.

For a prefect controllable Omni-directional movement the three independent drive units are mounted with the angle of 120° each other, which give Robotino® the
controllable degrees of freedom in all directions. These drive units are integrated in a laser welded steel chassis. The configuration feature of the three drive units are given below.

- A DC Dunker motor with of 3600 rpm
- Each motor consists a nominal torque of 3.8 Ncm.
- A Gear unit for each motor with a ratio of 4:1.
- The diameter of Omni-directional wheels is 80 mm.

In the figure 23 various parts of Robotino® Drive unit marked with numbers, (1) motor, (2) encoder, (3) omnidirectional wheel, (4) planetary gear, (5) toothed belt.

3.3 Kinect Sensor

Although the robot platform Robotino® comes with some useful sensors (discussed in section 3.1), for the implementation interest an additional RGB-D sensor is used. Kinect is a RGB-D based motion sensing input device product line by Microsoft. It was announced and demonstrated for Xbox 360 video game platform named as Project Natal in June 2009. The first-generation Kinect was first released on November 2010. Since then it attracts the attention of researchers and robotics developer community due to its immense possibility compare to traditional vision based system.

Figure 23 : Drive unit of Robotino®
Kinect was developed based on the software technology by Rare and on range camera technology by Prime Sense which developed a system that can interpret specific gestures, making completely hands-free control of electronic devices. Kinect features an RGB camera, depth sensor and multi-array microphone. The depth sensor consists of IR laser projector combined with CMOS sensor which captures 3D video data. The RGB camera is located between the IR projector and IR camera, which has no role in depth measurement. The specification of Kinect is given in below table.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Viewing Angle</td>
<td>43° vertical by 57° horizontal field of view</td>
</tr>
<tr>
<td>Vertical tilt range</td>
<td>±27°</td>
</tr>
<tr>
<td>Frame rate (depth and color stream)</td>
<td>30 frames per second (FPS)</td>
</tr>
<tr>
<td>Audio format</td>
<td>16-kHz, 24-bit mono pulse code modulation (PCM)</td>
</tr>
<tr>
<td>Audio input characteristics</td>
<td>A four-microphone array with 24-bit analog-to-digital converter (ADC) and Kinect-resident signal processing including acoustic echo cancellation and noise suppression</td>
</tr>
<tr>
<td>Accelerometer characteristics</td>
<td>A 2G/4G/8G accelerometer configured for the 2G range, with a 1° accuracy upper limit.</td>
</tr>
<tr>
<td>Nominal spatial range</td>
<td>640 x 480 (VGA)</td>
</tr>
<tr>
<td>Nominal spatial resolution (at 2m distance)</td>
<td>3 mm</td>
</tr>
<tr>
<td>Nominal depth range</td>
<td>0.8 m - 3.5 m</td>
</tr>
<tr>
<td>Nominal depth resolution (at 2m distance)</td>
<td>1 cm</td>
</tr>
<tr>
<td>Device connection type</td>
<td>USB (+ external power)</td>
</tr>
</tbody>
</table>

Table 2: Kinect Specification [69][70].
The Kinect sensor has a practical ranging limit of 1.2 - 3.5 m distance when used with the Xbox software. Also, the Kinect has built-in multi-array microphone, which contains four microphones for capturing sound. It is possible to record audio and the find location of source and the direction of the audio [70].

3.3.1 Functional mechanism Kinect depth sensor

PrimeSense, the developer of depth sensor has not published the technique of depth estimation. So, there has been a lot effort of reverse engineering to know some facts based on which the depth is measured. The IR projector projects IR light which falls on objects around it like a cloud of dots. Normally we cannot see the dots because that is projected in the Infra-Red color range.

![Figure 25: projecting a pattern of dots in infra-red from RGBD Sensor.](image)

But IR camera can see the pattern of dots. IR camera and RGB camera are essentially the same, except that IR camera captures image in the Infra-Red color range. After capturing images the IR camera sends of the distorted dot pattern into a depth sensor processor. The depth sensor processor estimates the depth of the pixels from the displacement of the dots. It works out on the pattern of the dots; the pattern is spread out on near objects and on the pattern, is dense on far objects. The result of the depth sensors processor is the depth map which can be read from the depth sensor into the computer or any other device.
Figure 26 shows a depth map image returned from the Kinect depth sensor. The specialty of this type of images are, it is grayscale image but with pixel depth information. Objects closure to the camera is lighter color and as the distance increase the color if getting darker. Based on this simple information, background foreground of image can be determined and object detection is easier with the depth map image.

3.3.2 Software and driver

Kinect was originally designed and released to be used in Microsoft Xbox 360 game console. No external driver was provided by the manufacturer. But as Kinect grabs attraction to the researchers and robotic development community, in November 2010, Adafruit Industries offered a bounty for an open-source driver for Kinect and in the same month Adafruit announced Hector Martin as the winner who produced a Linux driver for Kinect that allows using both RGB camera and depth sensitivity functions. It was named Open Kinect libfreenect which is the core library for accessing the Microsoft Kinect. Another open source framework, OpenNI framework was introduced by OpenNI organization. It provides for Kinect an application programming interface (API). There are 12 different nodes available in OpenNI framework for different functionalities. A tabular representation of the OpenNI nodes with the basic functionality is given in the table below.
<table>
<thead>
<tr>
<th>Device</th>
<th>The physical device representation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth</td>
<td>Provides depth map from generating the device sensor.</td>
</tr>
<tr>
<td>Image</td>
<td>Provide color image maps.</td>
</tr>
<tr>
<td>IR</td>
<td>Generates IR image maps.</td>
</tr>
<tr>
<td>Audio</td>
<td>Provides the audio stream generated by the microphone array.</td>
</tr>
<tr>
<td>Gestures</td>
<td>Create callback when specific gestures are identified.</td>
</tr>
<tr>
<td>Scene Analyzer</td>
<td>Analyzing result of scene, like foreground and background separation.</td>
</tr>
<tr>
<td>Hands</td>
<td>Creates callback event when hands point, gesture generated, changed or destroyed.</td>
</tr>
<tr>
<td>User</td>
<td>Represent user in 3D space.</td>
</tr>
<tr>
<td>Recorder</td>
<td>Records data.</td>
</tr>
<tr>
<td>Player</td>
<td>Play the recorded data.</td>
</tr>
<tr>
<td>Codec</td>
<td>Compression and decompression of data.</td>
</tr>
</tbody>
</table>

**Table 3: List of OpenNI nodes.**

Microsoft announced on February 21, 2011 for releasing non-commercial Kinect software development kit (SDK) for Windows and June 16, 2011 Windows software development kit (SDK) was released for Kinect which enables to create commercial or Windows Store apps, that support gesture and voice recognition by using C++, C#, Visual Basic, or any other .NET language. The integrated SDK toolkit also provides some basic applications resources to simplify and speed up application development.

For ROS, based implementation there are libraries available for Kinect driver. freencet_launch and openni_launch are packages for connecting RGBD sensors via ROS. After successfully launching the packages, the Kinect data (RGB image, depth-map, disparity image, etc) are published in ROS-topics and other nodes can subscribe those data from the published topics.

### 3.4 Point Cloud Library

The distance sensors of Robotino® and depth information from Kinect provide data in a special format called point cloud, which has a special type of data structure. The Point Cloud Library (PCL) is an open source library. PCL is used for large scale based 2D/3D image processing and point cloud data processing. PCL consisting of algorithms for point cloud processing tasks and 3D geometry processing specialized...
for three-dimensional computer vision. This library contains a huge number of modern algorithms for feature estimation, surface reconstruction, registration, model fitting, and segmentation, etc. The implementation code of the library is written in C++ and released under the BSD license, which is free for commercial and research purpose use. PCL is widely being used in various research and commercial projects. It is cross-platform, and thus compatible with different operating systems like windows, Linux, MacOS, Android/iOS, etc. To reduce the complexity of development, PCL is split into a series of smaller code libraries that can be compiled separately, which is called modularity. This modularity plays an important role for PCL distribution on different platforms with less computational constraints. Figure 27 shows a graphical representation about PCL code libraries and modularity.

![Figure 27: PCL modularity graph](image)

The original development of PCL started in March 2010. It started at Willow Garage and the project initially resided in a sub domain of Willow Garage. In March 2011, it was moved to a new website www.pointclouds.org and the first official release Version 1.0 of PCL was released two months later in May 2011.

The ROS has a package that works with the PCL libraries. It makes easier to use the PCL libraries function/algorithm within the ROS environment. For example, The Kinect sensor returns or publishes PointCloud2 data type messages in ROS environment, where the system might need some complex algorithm to use with the sensor information. The pcl_ros package works as a bridge between Kinect sensor message of PCL2 type and the ROS environment itself for application involved with 3D data. The available two type of point cloud structure are shown in the table below.
### PointCloud and PointCloud2 message type definition

<table>
<thead>
<tr>
<th>PointCloud</th>
<th>PointCloud2</th>
</tr>
</thead>
<tbody>
<tr>
<td>std_msgs/Header header</td>
<td>std_msgs/Header header</td>
</tr>
<tr>
<td>geometry_msgs/Point32[] points</td>
<td>uint32 height</td>
</tr>
<tr>
<td>sensor_msgs/ChannelFloat32[] channels</td>
<td>uint32 width</td>
</tr>
<tr>
<td></td>
<td>sensor_msgs/PointField[] fields</td>
</tr>
<tr>
<td></td>
<td>bool is_bigendian</td>
</tr>
<tr>
<td></td>
<td>uint32 point_step</td>
</tr>
<tr>
<td></td>
<td>uint32 row_step</td>
</tr>
<tr>
<td></td>
<td>uint8[] data</td>
</tr>
<tr>
<td></td>
<td>bool is_dense</td>
</tr>
</tbody>
</table>

Table 4: PointCloud and PointCloud2 message type definition [71].

Now, the distance sensor returns PointCloud type data and the Kinect sensors publish PointCloud2 type messages for 3D map data. The ROS package is very useful to handle these two data types.

### 3.5 OpenCV

Open Source Computer Vision Library (OpenCV) is very popular open source library specially for image processing. OpenCV is specialized on complex computation for computer vision and machine learning based algorithm and optimized for real time based implementation. The main intension of OpenCV was to provide a common platform and infrastructure for computer vision applications and to accelerate the use of machine perception in the embedded application related commercial products. OpenCV is BSD-licensed, which is free for commercial and research purpose use. OpenCV library has more than 2500 optimized modern algorithms, including comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. The OpenCV library has thousands of useful algorithms for complex image processing which can reduce the time to development of application and implementation. Algorithms like image comparison, image matching, face detection, red eye removal, image resizing, identify and tracking of objects, extracting 3D models of objects, classify actions in videos are provided in a very optimized way in the library.

To work with OpenCV in collaboration with ROS, already there are some libraries available. OpenCV2 is the official version supported on ROS Indigo and Jade. `vision_opencv` is the ROS library stack which provides packaging of the most popular OpenCV functionalities. It has several useful packages. In the next sub sections those are described briefly.
3.5.1 cv_bridge

This contains CvBridge, which bridges between ROS messages and OpenCV by converting between ROS Image messages and OpenCV images.

![Diagram](image)

Figure 28: cv_bridge between ROS and OpenCV.

ROS has data structures of its own type, on the other hand OpenCV use some of the well-known generalized data structure. This package mainly converts the data types from ROS to OpenCV or vice versa.

3.5.2 image_geometry

This package is a collection of useful functions for dealing with image and pixel geometry. It contains camera model classes written in C++ and Python which simplify interpreting with images geometrically using the calibration parameters from sensor_msgs/CameraInfo.
4 Implementation

In previous sections mentioned all the research ideas, tools, technologies are the base for this study. This is obvious that already there are a good number of solutions available for obstacle avoidance robot. Some algorithms are very well performed in consideration with their result but on the other hand, some of them are very complex or not suitable for real time based solution or might not fit for a simple design based implementation. In this thesis, simple obstacle detection and efficiently avoiding the collision for an indoor mobile robot is considered with the focus of detecting moving obstacle and its direction of movement. The approach is chosen with minimal computational effort to find the most common obstacles (i.e.: humans, chairs, tables, doors, etc.) in an indoor environment. The obstacles can be divided mainly into two classes, moving and stationary objects. As a target platform Festo Robotino® mobile robot is chosen and the feature of this robot is already discussed in previous chapter. The implementation is done on Ubuntu Linux based system, and ROS. As for coding purpose C programming language is used as ROS, OpenCV, PCL is supported for C language based implementation. Most of the algorithm implementation is done utilizing OpenCV functions. From mapping, navigation to obstacle detection and avoidance, every necessary step to design and implement the obstacle avoidance system for indoor mobile robot is given below sections.

4.1 Process flow design

Robotino® is used as a robot platform with additional sensor Kinect and ROS based implementation has been considered for mapping, navigation, connecting Kinect and controlling the robot altogether. As discussed in section 3.1, ROS communicates with different nodes via connected network and the connection is established first with Master node. Once a node is registered and recognized by master node, it can then communicate with all the other nodes which are also registered with the same master node. In this implementation master node, along with mapping, navigation and several other nodes will be hosted from a laptop computer (server machine), which is connected to Robotino® via a wireless network. The Kinect driver node will launch in the Robotino® itself. There will be a node named robotino_node which will also be in the computer and it will be responsible for controlling, read, and write from different sensors and topics from Robotino®. So, once every node is registered by a single master node, then every node can communicate each other, regardless the node is running from the robot, server machine. This is how multiple robot or machine can be communicating each other. Figure 29 shows the architecture diagram of the communication pattern. The robotino_node is responsible to bring up the robot and
all its sensors available for other nodes and programs to communicate. The node in the robot Kinect_driver brings up the Kinect sensor to ROS. Mapping node creates the map of the navigation world with the help from the sensors. The navigation node navigates the robot from a given source to destination and avoids the obstacle. It takes continuous feedback from the sensors and takes decision. The detail processes of the node are described in the next sections.

![Diagram of Process Architecture](image)

**Figure 29**: Process Architecture.

### 4.2 Connecting to Robotino®

As the process architecture shows that the robot needs to connect with Robotino®, ROS communication is used as a bridge for connecting between the two devices. The Robotino® comes with a Linux Ubuntu operating system. In this implementation, the server machine is also configured with Ubuntu 14.4 version. There are several versions of ROS is available. During the configuration time the latest available stable version was ROS-Indigo. Both the computer and Robotino® installed with ROS Indigo. First, the master node needs to be up and running. In the proposed design the master node is run in the server computer. The server computer and robot need to be in the same network so that they can be connected through wireless network. The robotino_node is a node which interfaces the Robotino® API2 with ROS. It activates all the sensors attached with the Robotino and publishes the data. These data are published on different topics. It implements a few services too. This node also subscribes to a few topics to receive set of values. The connection and launching commands are given below in figure 30.
In the above image, the commands are executed from the server machine. The first line initializes the master node and it resides in the server machine. The second line initializes the robotino_node which is also in server machine but connected to the Robotino as the network address of Robotino® is given as a parameter. The Robotino® must be turned on and the wireless network should be configured.

4.3 Mapping and Navigation

For the implementation purpose of the object detection module, the robot mapping and navigation must implement on the robot platform. Mapping is a must for navigation in a known environment. The map could be either provided by drawing manually or can be created using the sensors which generates data for a map. A robot is moved manually to every corner of the location and the sensors continuously take data to create a map. SLAM based mapping are very useful for navigation of robot as the map contains local information more accurately from the environment. ROS have a SLAM based mapping package `slam_gmapping`, which contains a ROS wrapper for OpenSlam’s Gmapping. This package provides a laser-based SLAM. It can create a 2D grid map using laser and odometry pose data collected from mobile robot sensors. To use `slam_gmapping`, a mobile robot is needed which provides odometry data and equipped with a laser range-finder. The `slam_gmapping` node combines the scanned data and odometry information and attempts to transform into the odom tf frame. The problem with existing platform is, Robotino® has odometry information but it is not equipped with a laser range finder by default. To overcome the problem, Kinect sensor will be used to scan the depth data of sensor_msgs/PointCloud2 type and ROS package `depthimage_to_laserscan` will be used to convert the Kinect depth data into a sensor_msgs/LaserScan typed data. The type description of PointCloud and laser scan is given in the below table.

<table>
<thead>
<tr>
<th>Sensor_msgs/PointCloud2</th>
<th>Sensor_msgs/LaserScan</th>
</tr>
</thead>
<tbody>
<tr>
<td>std_msgs/Header header</td>
<td>std_msgs/Header header</td>
</tr>
<tr>
<td>float32 angle_min</td>
<td>uint32 height</td>
</tr>
<tr>
<td>float32 angle_max</td>
<td>uint32 width</td>
</tr>
<tr>
<td>float32 angle_increment</td>
<td>sensor_msgs/PointField[] fields</td>
</tr>
<tr>
<td>float32 time_increment</td>
<td>bool is_bigendian</td>
</tr>
<tr>
<td>float32 scan_time</td>
<td>uint32 point_step</td>
</tr>
</tbody>
</table>
float32 range_min  
float32 range_max  
float32[] ranges  
float32[] intensities  

uint32 row_step  
uint8[] data  
bool is_dense

| Table 5: PointCloud2 and LaserScan message type definition [71]. |
|-------------------|-----------------|
| float32 range_min | uint32 row_step |
| float32 range_max | uint8[] data    |
| float32[] ranges  | bool is_dense   |
| float32[] intensities |             |

Figure 31, next shows the process of converting depth image data to laser scan data type using the ROS package `depthimage_to_laserscan`, (a) Real scene (b) scanned depth image by Kinect (c) sensor message laser scan type projected on top based on depth data. (d) Top down view of the laser scan.

![Figure 31: Depth image to lasercan conversion process. [51]](image_url)

The overall mapping process is done in two steps. In the first step, the robot moves and scans data. The scanned data is stored in a file called BAG file.
In the second step, when the BAG file is generated then the `slam_gmapping` command creates the map by playing the BAG file which was generated in the previous step. The commands and instructions to generate map are given in the figure 33 below.
There are several options to move the robot manually. In this case the ROS node `robotino_teleop` is used.

The navigation part makes the robot move autonomously from source point to destination. For known environment (in this case it is mapped environment), the autonomous robot can freely navigate within the map. There is already a lot of robot navigation algorithm available and some of them are discussed previously in section 2.1. There are several navigation packages available for autonomous robot navigation. `teb_local_planner` is a navigation and path planning package implemented for ROS based on a very modern approach for path planning named Time Elastic Band. This approach is selected because it is an online trajectory planner for autonomous mobile robot which allows finding multiple paths and can also avoid collision with defined obstacles. The multiple path planning features makes it more reliable in compare with another navigation module.

Using the ROS visualization software rviz, without connecting with the robot the navigation module can be tested by simulation. To move the robot, the `robotino_node` must be launched along with the navigation node. Figure 34 is a screen shot of using rviz for simulation of the navigation.
Figure 34: Navigation simulation using rviz.

The green line is the global path generated using navigation algorithm and it ends at the destination point. The blue circle points the robot. There is a semi-transparent white square around the robot which covers the area of the local path planning. The red line is the local trajectory which interacts with the obstacles. Once all the nodes are active, from the rviz tool a user input can be set as source and destination point. The navigation module immediately starts calculating the distance and paths and starts moving towards the destination. During its movement it will continuously sense the environment from the sensors and if it finds any object within the range of the sensors, the information is forwarded to the local planner. The local planner calculates the alternative trajectory to avoid the obstacle. If there is an alternative safe path which can avoid the obstacle then it moves otherwise it stops until the obstacle is not removed. Primarily the robot is supposed to follow the global path and if there is any obstacle within the range of local path planning, then the local path modifies the path to avoid the obstacle. Using the modified local path the robot avoids the obstacle and it continues with the global path again.

4.4 Obstacle Detection

Once the navigation module is working perfectly, it assures the robot can navigate within the map from a source point to the destination point. If there are no changes in real scenario with the generated map, then the robot is supposed to work smoothly. But it is more often that there are some unexpected objects on the pathway of the
robot. The objects that come around the navigation path regardless static or dynamic are concerned as obstacle for the robot. The navigation algorithm provided by the ROS is smart enough for the robot to avoid those obstacles in an optimal way. But the navigation module needs to know the information by sensing from the environment. If there are no sensors activated, then the navigation module will not be able to work with the dynamic changes of the environment. The distance sensors, embedded with Robotino® will be used to detect the obstacles within the range of 40cm around the robot base. But it does not cover the height from the ground more than 10cm. Also, to cover a larger distance Kinect sensor will be used which has a standard 3D sensing capability of 0.7m. The bumper sensor attached with the base will also secure the robot in case there is collision with something undetected by the Kinect and distance sensor. The overall view of the obstacle detection architecture is given in the figure 35 below.

Figure 35: Efficiently obstacle detection with multiple sensors.

To reduce the complexity and data load on the obstacle topic of navigation module, obstacle detected from any of the sensors will pass through only one channel. Different types of sensors return different type of response messages. All the feedback will be collected separately but will be processed via a single node and only the obstacle information will be passing through the channel. Also, using the 3D depth sensors, it is possible to detect the direction of the moving obstacle. This information is very useful when the navigation module cannot decide the path to avoid. In this implementation, the data from the sensors will be passed to navigation
module and if any moving obstacle is detected in the direction towards robot, the robot is stopped immediately.

As discussed in earlier chapter about ROS communication, the sensor data is published to a topic and other nodes can subscribe to the published data from the topic.

![Diagram](image)

Figure 36: Obstacle detection process.

In the figure 36 above, the whole ROS communication process for obstacle detection is described. Below section will describe the implementation process of detecting obstacle using different sensors.

### 4.4.1 Obstacle detection by Distance Sensors

To activate the distance sensors, no other extra driver or software is needed. The roboitno_node activates the distance sensors. It starts sensing and publishes to the ROS topic distance_sensors. The distance_sensor_listener is a subscriber node which always listens to the topic /distance_sensors. It calculates the distance of any particle within the range and if it finds any obstacle it publishes a Boolean type message on the topic /ir_obstacle_detect.
4.4.2 Obstacle Detection by Kinect

The depth_point_listener is also a subscriber node which listens to the topic /camera/depth/points. It calculates the distance from the depth image and if there is any object within the range then it also sends similar Boolean type message to the topic /Kinect_obstacle_detect. The subscribed topic /camera/depth/points return value in PointCloud2 data format. It returns data for each frame. Every point has x, y, z coordinate value. To understand the calculation process of the distance measurement, figure 37 below can be taken as a reference.

From the above figure, consider that the returned data frame from the depth sensors is the XY plane. C is the center of the plane and P is any point on the plane. From the image 37, Z is the perpendicular distance from the plane to the camera. The perpendicular drawn from the camera to the plan hits at center of the XY plane, where X is the horizontal and Y is the vertical axis. C is Z meter away from the center of camera. Now Z is the perpendicular distance from the point to the camera plan but to find the actual distance d from the point to the camera, the hypotenuse hypot(p->z, p->x) need to calculate, in equation1 the process of hypotenuse calculation is given. Angle convention to the left of the camera is 0 degree and to the right is 180 degree (deals with X axis), left 0-----]-----180 right. Vertically the bottom of the camera is 0 degree and to the top is 180 degree (deals with Y axis). So, angle made by the point horizontally,

$$\text{min\_angle\_radX} = \text{atan2}(p.Z, p.X) \ldots \ldots (7)$$

Equation 7: Horizontal angle.

And angle made by the point vertically,
\[ \text{min_angle_radY} = \text{atan2}(p.Z, p.Y) \cdots (8) \]

Equation 8: Vertical angle.

This angle calculation is useful to filter the region of interest. The points which are too high above or in the ground is not considered calculating for obstacle detection.

### 4.4.3 Detect Moving Obstacle

The moving_obstacle_detection node subscribes the message from the topic `/Kinect_obstacle_detect` and if there is anything within the range it starts calculating the process to find whether if there is anything moving or stationary. It subscribes to the topic `/camera/depth/image_raw`, which publishes depth image. Depth image consist of data of each image points and it is a grey scale image which is light weight compare to color image. The goal is to use Kinect sensor in compare with normal camera is, this depth image reduces the computational complexity to calculate the object movement detection from a colored 2d image. The figure 38 below shows the depth image side by side with the actual image taken from a webcam attached with Robotino®.

![Real image from web cam (Left) and Depth image from Kinect (Right).](image)

The disparity image from Kinect also returns the depth information with additional color mapped with the distance. In the disparity image, the closure the pixel towards camera becomes red color and as the distance increase the color become gradually grey color. Figure 39 below is the disparity image taken from the Kinect.
As this process is a bit more computational and to make it more optimized, this process only works when there is a response from the Kinect sensors that there is something within the range. Otherwise, this process will do nothing. After processing, it creates a Boolean result that if the obstacle is moving or not and then the result data is published to the topic `/moving_detection`. The pseudo code for detecting moving obstacle is given on Figure 40.

When the node gets notification of existence of any object within the range, at first the node sends command to the robot to stop immediately. This is done due to safety reason. As the main concern of this study is to safety and collision avoidance, the robot is stopped unless it gets a solid decision about further movement. Once the robot is stopped, it waits for three consecutive frames from the topic `/camera/depth/image_raw`, which gives depth image frames. The real data that obtained from the sensors contains a lot of noise and disturbance. So, before implementing the actual theory of the algorithm adds some smoothing and noise filtering mechanism. Otherwise, the result will not go according to the theoretical concept. When it gets all the three frames, it applies Gaussian blur method to all three images to make it smoother by filtering.
In the given algorithm, above, line number 9 and 10 are mentioned to smooth the noisy image. There are several types of blurring algorithms like median blur, bilateral blur, etc. But Gaussian blur is simple and most commonly used for image processing. OpenCV provides an implementation of this algorithm function. The function is specified with a Gaussian kernel, the filtering process is done by iterating each point in the input array with that kernel and then adding them up all for producing the output array. The OpenCV Gaussian blur sample input and output is given below.
Next it applies the three-frame difference method on the blurred smooth images and in the algorithm line 12, 13 and 14 is mentioned for the frame difference. In section 2.2.2 it has been discussed with reference of equation 4, equation 5 and figure 6. To find out the image frame difference OpenCV method absdiff() is used. It calculates the absolute difference between two arrays and returns the result array. Theoretically the difference of two identical arrays will be zero. If there is anything moving, two consecutive frames will have some difference. But as the real data are two noisy, even after smoothing the images it will consists some noise and will give some difference. So directly the result array for frame difference we cannot use as a decision maker for detecting moving elements. The next Figure shows the output of frame difference images taken from a still camera position and nothing was moving in front of camera but the output shows that still there are some pixels’ differences and the two image are not identical.

The frame difference image of two consecutive frames with some moving object is shown in the next figure, where a movement of human hand makes a clear impact in the output image.
However, in both the cases above the camera was in stationary position. Only the object in front of the camera was moving in the figure above. In the real implementation, the camera will be attached with the robot, so the camera will also be in moving position. It is not possible to imply the frame difference method when the camera is also moving. For this reason, the robot is stopped immediately when it senses something in front of the camera within the range and it is mentioned in the above algorithm in line number 3. The difference images (figure 42, figure 43) contain noisy data and in the next step there will be some pressing to remove efficiently those noise data.

The frame difference image gives some visible information about moving object. But several factors like lights or shades can make big impact on frame difference and make it noisy. In the frame difference image, the background or static part of the image is clearly black, because the difference of the same pixel value is zero. Regions that contain motion are much lighter, although on that region everything is not moving. To get rid of this, a threshold value is applied, which is similar to foreground and background detection approach. Line number 17 of the algorithm is mentioned about applying the threshold value over the frame difference image. OpenCV function threshold() is used with the type binary threshold as it serves the purpose. The binary threshold can be expressed with the below equation.

\[
\text{dst}(x, y) = \begin{cases} 
\text{maxval} & \text{if } \text{src}(x, y) > \text{thresh} \\
\text{minval} & \text{otherwise}
\end{cases}
\quad \ldots \quad (9)
\]

Equation 9: Binary Threshold.

For the implementation maxval is chosen 255, minval is 0 and thresh is set to 10. This threshold value can be tuned (increased or decreased) depending on where and how the implementation will be done. An image of binary threshold operation is given in figure 44, where the difference image is the output of moving hand. Although
several noise filter operations have been applied so far, there still exist some unwanted noise data in the threshold image which needs to eliminate.

![Image](image1.png)

**Figure 44**: Frame difference image (Left), Threshold Image (Right).

To reduce the noises, morphological analysis on the threshold image will be applied. There are different types of morphological operations already implemented by OpenCV library. A simple dilation operation works well on the binary images captured in an indoor environment. Other morphological analysis options like erosion, opening, closing, etc can also be replaced by dilation. For the experimental purpose, dilation and opening operation was used separately. And the result was quite similar for indoor environment. In the mentioned algorithm line number 20 calls the OpenCV dilate function and return the resultant image. Next figure shows the result output image of morphological analysis operation (dilation) done over the threshold image obtain previous.

![Image](image2.png)

**Figure 45**: Threshold binary image (Left), Output of morphological analysis (Right).

After getting the dilation image the function findContours will return the set of contours available in the image and the function drawContours will draw the contours
to an image canvas. Contours are simply a curve line which is formed by joining all the continuous points in an image having similarities like color or intensity. It is very useful tool for shape analysis, object detection and recognition from an image. Finding contour method is implemented as a function by OpenCV using the algorithm proposed by Suzuki, S. and Abe, K. [54]. It will create regions by joining similar type of pixels based on the algorithm. The noisy data are very much scattered and it will create a very small region after the dilation, on the other hand the area of the moving object will generate region almost closer to the size of the moving object. For better accuracy, it is recommended to use binary images. Therefore, before finding contours, threshold and morphological analysis operation already applied. Also for using OpenCV findContours method this is to mention that, finding contours is like finding white object from black background. To eliminate the contours those are generated for noisy data the drawContours will only generate the contours having a minimum area which is defined or set from a statistical data analysis. Now for the statistical analysis part normality test 68–95–99.7 rule has been applied. The camera is moved several times in the environment and more than 10 thousand frame data are captured in different light condition (natural light, artificial light). From those frames generated contoured are stored in a flat file and then statistical analysis has been done simply using by the spread sheet analysis tools (i.e.: Microsoft excel). The average size was obtained a little above 1000, which is stored as a normal_value. In the line number 22 of the algorithm, the variable contour_list gets the list of all the contours using the method findContours() and from line 24 to 26 it iterates over all the contours and calculate the area of each contours. If the contour area is greater than the normal value, the system draw the contour and sets the decision that an object is moving in front of the camera. The figure next shows the contour drawn from the image obtained after morphological analysis.

![Figure 46: From Morphological dilation image (Left) to draw contour image (Right).](image-url)
4.4.4 Combined decision for obstacle detection

The node combined_decission subscribed all the result from the three nodes. It creates a combinational logic operation for deciding and if any of the sensors detect any kind of object it publish a message over the topic cmd_vel to stop the robot immediately.

![Combined Decision Diagram](image)

It also publishes the obstacle information to the topic/move_base/TebLocalPlannerROS/obstacles, which is subscribed by the navigation module local planner to make dynamic and collision free trajectory planning and this topic accepts data in Obstacles type format. This type is defined by teb_local_planner module for ROS and the type information is given in the table below.

<table>
<thead>
<tr>
<th>teb_local_planner/ObstacleMsg</th>
</tr>
</thead>
<tbody>
<tr>
<td>std_msgs/Header header</td>
</tr>
<tr>
<td>geometry_msgs/PolygonStamped[] obstacles</td>
</tr>
<tr>
<td>uint32[] ids</td>
</tr>
<tr>
<td>geometry_msgs/QuaternionStamped[] orientations</td>
</tr>
<tr>
<td>geometry_msgs/TwistWithCovariance[] velocities</td>
</tr>
</tbody>
</table>

Table 6: Obstacle message type Definition.

After the moving obstacle is detected by any contour, OpenCV convex hull method is applied upon the contour. The image below is taken from OpenCV documentation page which shows, how it looks like after applying convex hull over an image of a hand.
Thus, a shape or polygon is generated covering the area of that contour. Then the convex hull shape is converted to an obstacle type message. From the above table of obstacle message type definition, the polygon or shape is given with all the geometry message information, so the navigation module can easily identify the current moving position and try to avoid possible collision.
5 Evaluation

5.1 Implementation Result

The overall system is implemented on Robotino® robot platform and evaluated with several conditions and constraints. As for Kinect based detection light and distance is a vital issue. The test evaluation is done for several types of objects, moving or static things and on several lights conditions like day time, night time, with or without artificial light, natural light, etc. The best accuracy for distance sensor data comes within the distance range of min 5 to 35 cm range. Within this range it detects objects with best accuracy in our test.

Figure 49, shows some images for sample observation. On the top left (a) wooden block is put in front the sensors, top right (b) is a transparent glass bottle to observe obstacle detection and in the bottom (c) image a hanging cable taken as sample to observe.
For Kinect based obstacle detection, the best accuracy range covers from 0.8m to 1.5m although it can detect data between the mentioned ranges. For the IR sensors, the test result for different type of sample are listed in the table below.

<table>
<thead>
<tr>
<th>IR Sensors</th>
<th>Obstacle Detection Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test sample</td>
<td>Natural/Artificial Light</td>
</tr>
<tr>
<td>Metal/Shiny Object</td>
<td>Yes</td>
</tr>
<tr>
<td>Concrete Wall</td>
<td>Yes</td>
</tr>
<tr>
<td>Paper box</td>
<td>Yes</td>
</tr>
<tr>
<td>Transparent Bottle</td>
<td>Yes</td>
</tr>
<tr>
<td>Transparent Plastic</td>
<td>Yes</td>
</tr>
<tr>
<td>Moving Object</td>
<td>Yes</td>
</tr>
<tr>
<td>Hanging Object (cable/wires)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 7: Test objects and results for IR sensor

Kinect based object detection result on sample objects are listed in the table below. There are some cases where the result output has no consistency. It depends on the environmental effect like light reflection, darkness, etc.

<table>
<thead>
<tr>
<th>Kinect Sensors</th>
<th>Obstacle Detection Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test sample</td>
<td>Natural/Artificial Light</td>
</tr>
<tr>
<td>Metal/Shiny Object</td>
<td>Environment Sensitive</td>
</tr>
<tr>
<td>Concrete Wall</td>
<td>Yes</td>
</tr>
<tr>
<td>Paper box</td>
<td>Yes</td>
</tr>
<tr>
<td>Transparent Bottle</td>
<td>Yes</td>
</tr>
<tr>
<td>Transparent Glass window</td>
<td>No</td>
</tr>
<tr>
<td>Transparent Plastic</td>
<td>Yes</td>
</tr>
<tr>
<td>Moving Object</td>
<td>Yes</td>
</tr>
<tr>
<td>Hanging Object (cable/wires)</td>
<td>Yes</td>
</tr>
<tr>
<td>Corner/Sharp edges</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 8: Test Objects and results for Kinect Sensors

For moving obstacle or movement detection from the Kinect sensors, the system has been tested with different types of moving obstacle like human, chair, doors, toys etc. Moving with average walking speed or higher than that is well detected but objects that are moving too slowly are sometime not detectable (if it is slow enough to change within three sequential frame). Also, the moving object detection response
time is comparatively slower when the Kinect object detection system is integrated with Robotino® and navigation module. The sample test combination for movement detection from the Kinect sensors is given in the table below.

<table>
<thead>
<tr>
<th>Kinect Sensors</th>
<th>Movement Detection Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test sample</td>
<td>Natural/Artificial Light</td>
</tr>
<tr>
<td>Metal/Shiny Object</td>
<td>Environment Sensitive</td>
</tr>
<tr>
<td>Human movement</td>
<td>Yes</td>
</tr>
<tr>
<td>Moving Chairs</td>
<td>Yes</td>
</tr>
<tr>
<td>Transparent Bottle</td>
<td>Yes</td>
</tr>
<tr>
<td>Very thin object</td>
<td>No</td>
</tr>
<tr>
<td>Environment Sensitive</td>
<td></td>
</tr>
</tbody>
</table>

Table 9: Test results for Movement detection by Kinect.

For obstacle detection using distance sensors and Kinect and for detecting moving obstacles, the experiment results are observed around 150 times with different combination and during observation the distance range was maintained. It happened that sometime for the same condition and same sample data, the result output is different due to environmental sensitivity like lights and shades. The success rate for each case is described in the table below.

<table>
<thead>
<tr>
<th>Sample observation (times)</th>
<th>Success (times)</th>
<th>Success Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obstacle Detection – IR Sensors</td>
<td>150</td>
<td>150</td>
</tr>
</tbody>
</table>

Table 10: Test Observation Success Rate for IR Sensors

Table 10 is showing the result of the obstacle detection from distance sensors. The success rate is 100% as for 150 test observation the sensors detected obstacles every time successfully. Next table shows the success rate of obstacle detection using Kinect depth image. The success rate is 86% for 150 test observation. The test objects were mainly moving objects, human, chair, tables, etc. The experimental observation was done using daylight, dark room, and electrical lamp to see the environmental effect.
The next, table 12 show the success rate of detecting moving obstacles using Kinect. The output result was quite good but for thin objects like hanging electrical cable does not make any impact for detecting as a moving object. Also, it has an environmental sensitivity impact as mentioned in table 8 and table 9.

<table>
<thead>
<tr>
<th>Sample observation (times)</th>
<th>Success (times)</th>
<th>Success Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obstacle Detection – Kinect</td>
<td>150</td>
<td>129</td>
</tr>
</tbody>
</table>

Table 11 : Test Observation Success Rate for Kinect

<table>
<thead>
<tr>
<th>Sample observation (times)</th>
<th>Success (times)</th>
<th>Success Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Movement Detection – Kinect</td>
<td>150</td>
<td>112</td>
</tr>
</tbody>
</table>

Table 12 : Test Observation Success Rate for Moving Obstacle detection

Some images from moving obstacle detection process using Kinect are given below.

![Image 1](image1.png)

![Image 2](image2.png)

![Image 3](image3.png)

Figure 50 : Kinect Movement Detection Result with static Obstacle

In the figure 50, from top left first two images are two consecutive frames and the third one is the frame difference. Bottom left image shows the threshold image and middle one is the result of morphologic analysis (dilate) on threshold image. Bottom right image is the ultimate result of movement detection of obstacle represented by contour drawing. In the figure 51 below, the six images are same result as the
previous one. But in figure 5.0 there is no moving object so in the last frame there is nothing to show; on the other hand figure 5.1 has a contour output at the last image which indicates the result that there is a moving object in front of the Kinect.

Figure 5.1: Kinect Movement Detection Result with moving Obstacle

5.2 Further Scope

This implementation finds obstacle and sends the information to the navigation module to find an alternative way to avoid the collision. If there is no other alternative path is found the robot stops immediately and waits until there is a free path available. Also for any kind of object detection the robot is stopped first for safety reason. But if the obstacle is moving towards the robot then this implementation is not able to take any protective action (like moving backwards). It is possible to get the moving direction of the obstacle using the depth map data. As a further scope, an improved algorithm based implementation can be proposed for taking any protective action to avoid the collision from the objects that are moving towards the robot.

From the observation of the implementation result in previous section, the conclusion is that the system can detect obstacle in different environmental constraint using IR sensor and Kinect sensors within the expected distance range. But the ratio of showing output is not 100%. The scrutinized reason behind that issue was mainly the communication time between nodes and sensor data from subscriber/publishers. As discussed in the implementation chapter, the master_node is implemented in the server machine with several other nodes but the Kinect sensor node was implemented in the Robotino®. The other sensors data like IR sensors, odometry information are passed through the wireless network from Robotino® to the server.
machine. Also, the data publishing from the nodes which reside in server machine are passing through the network to the Robotino®. These huge server-Robotino® communication makes a propagation delay. If the network bandwidth is not too high to deal with the data, then the object detection module will not be able to respond in time. One quick solution for this could be to implement all the nodes into the Robotino®. The embedded PC with default configuration of the Robotino® (Figure 17) is powerful enough to handle all the nodes for a single robot based implementation. Another observation with IR sensor was, in the current implementation all the nine sensors were activated for continuous data subscription and publication. While robot moves forward, it is enough to enable only the sensor 1, 2, 9 (figure 19) attached to the front part of the Robotino® base.

As the implementation was done in C programming language using the ROS and OpenCV library functions, it can be re-structure in more generalized way to publish as a ROS library for moving obstacle detection using Kinect.
6 Conclusion

This thesis work investigates the strategies of object detection using Kinect and other sensors equipped in robot by implementing on Robotino®. The aim of the approach was to evaluate the implementation on Robotino® to minimize the collision risk for more safety on indoor mobile robot navigation. The advantage of the implementation strategy was the uses of existing tools and algorithms from previous research and combining different type of sensor utilization which brings a wider range to sense information from the surrounding and jumps for a decision. The distance sensors detect obstacle in short distance and the Kinect detects obstacles where the distance sensors range cannot be reach. The Kinect also detects moving obstacle, which give added dimension for safety of the robot. As the primary purpose of this thesis was to detect obstacle rather than finding any kind of pattern of the objects (i.e.: face detection, human detection, etc.) or tracking the objects, the simplest possible strategy was approached and it works.

ROS is a platform which brings a wide range of flexibility for using several tools and libraries. Tools like OpenCV, PCL brings optimized implementation of the state-of-the-art algorithms which makes life easier. In this implementation, several functions have been used from OpenCV library. Although this implementation was expected to outperform theoretically but in practical the success rate is not 100%. The size and shape of the object, movement speed, lights, communication speed, etc. play an important role in its delectability by vision. It is entirely possible to enhance the performance of vision-based collision avoidance using superior image processing strategies. Future research will address this issue.
7 Appendix

For the implementation purpose of this thesis couple of the methods and functionalities described in the chapter four was written using the programming language C/C++, ROS, OpenCV library. Some important code snippets are attached here in this section.

7.1 Obstacle detection by IR Sensors

The code for the node obstacle detection by IR sensor is given below.

```c
#include <ros/ros.h>
#include <std_msgs/Bool.h>
#include <pcl_ros/point_cloud.h>
#include <pcl/point_types.h>
#include <boost/foreach.hpp>
#include <pcl/filters/passthrough.h>
#include <stdio.h>
#include <math.h>
#include <boost/shared_ptr.hpp>
#include <sensor_msgs/PointCloud.h>
#include <std_msgs/String.h>

/**
 * This tutorial demonstrates simple receipt of messages over the ROS system.
 */
// %Tag(CALLBACK)%
using namespace std;
ros::Publisher pub, dec_pub;
void distance_sensor_Callback(const sensor_msgs::PointCloudConstPtr& msg) //rasel
{
    double C_OFFSET = 0.612046 - 0.40;
    int i = 0;
    double minDistance = 0.0;
    double min_angle_radx = 0.0;
    double min_angle_rady = 0.0;
    double xX = 0.0, yY = 0.0, zZ = 0.0;
    int count = 0;

    for (i = 0; i < msg->points.size(); i++) {
        ROS_INFO("...........
```
ROS_INFO("Y : %f", msg->point[i].y);
ROS_INFO("Z : %f", msg->point[i].z);

minDistance=sqrt(pow(msg->point[i].x, 2)+pow(msg->point[i].y, 2)+pow(msg->point[i].z, 2))-C_OFFSET;

ROS_INFO("Distance= %f \n", minDistance);
std_msgs::Bool msg_decission;
msg_decission.data = false;
if(minDistance<0.3)
{
    msg_decission.data = true;
    pub.publish(msg_decission);
    break;
}
else
{
    msg_decission.data = false;
    pub.publish(msg_decission);
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "listener_distance_sensor");
    ros::NodeHandle nh;
    ros::Subscriber sub = nh.subscribe("/distance_sensors", 1, distance_sensor_Callback);
    pub = nh.advertise<std_msgs::Bool> ("/ir_obstacle_Detect", 1);
    ros::spin();
    return 0;
}

7.2 Obstacle detection by Kinect Sensors

The code for the node obstacle detection by Kinect depth image information is given below.

#include <ros/rosh>
#include <std_msgs/Bool.h>
#include "std_msgs/String.h"
#include "geometry_msgs/Twist.h"
#include <sensor_msgs/LaserScan.h>
#include <pcl_ros/point_cloud.h>
#include <pcl/point_types.h>
#include <boost/foreach.hpp>
#include <pcl/point_types.h>
#include <pcl/filters/passthrough.h>
#include <stdio.h>

using namespace::std;
std_msgs::Bool check_moving_obj , obstacle_found, is_moving_obs, break_loop;
int last_obs_found=0;
ros::Publisher pub, dec_pub;

typedef pcl::PointCloud<pcl::PointXYZ> PointCloud;

PointCloud pcl_previous, pcl_current;

double scale_linear_, scale_angular_;

void scan_callback(const sensor_msgs::LaserScan::ConstPtr& scan_in){
    ROS_INFO("Laser Scan Data... range_max = %f \n", scan_in->range_max);
}

void callback(const PointCloud::ConstPtr& msg){
    break_loop.data = false;

    std_msgs::Bool msg_decission;
    msg_decission.data = false;
    is_moving_obs.data = false;

    double minDistance=0.0;
    double maxDistance=0.0;
    double min_angle_radx=0.0;
    double min_angle_rady=0.0;
    double xX=0.0,yY=0.0,zZ=0.0;
    int count=0;

    // Angles are calculated in radians and can convert to degree by multiplying it with 180/pi
    cout<<"---------------------msg size is ;"<<msg->points.size()<<"\n";
    BOOST_FOREACH (const pcl::PointXYZ& pt, msg->points)
    {
        if(break_loop.data == true)
        {
            break;
        }
        if(atan2(pt.z, pt.y)*((180/3.14159265358979323846)>80.00)
{ // atan2(z,y) = arctan(z/y) if z>0;
// truncating points with less that 80 degree vertical angle
// because the point formed could be ground.
if(count==0)
{
    // initializing the first point read as minimum distance point
    maxDistance=hypot(pt.z, pt.x);
    minDistance=hypot(pt.z, pt.x);
    min_angle_radx=atan2(pt.z,pt.x);
    min_angle_rady=atan2(pt.z, pt.y);
    xX=pt.x;
    yY=pt.y;
    zZ=pt.z;
}
else if(hypot(pt.z, pt.x)<minDistance)
{
    // keep updating the minimum Distant point
    minDistance=hypot(pt.z, pt.x);
    min_angle_radx=atan2(pt.z,pt.x);
    min_angle_rady=atan2(pt.z, pt.y);
    xX=pt.x;
    yY=pt.y;
    zZ=pt.z;
    if(minDistance<0.6)
    {
        msg_decission.data = true;
        geometry_msgs::Twist cmd_vel_msg;
        double vel_x, vel_y, vel_omega;
        vel_x = 0.0;
        vel_y = 0.0;
        vel_omega = 0.0;
        cmd_vel_msg.linear.x = scale_linear_ * vel_x;
        cmd_vel_msg.linear.y = scale_linear_ * vel_y;
        cmd_vel_msg.angular.z = scale_angular_ * vel_omega;
        dec_pub.publish(cmd_vel_msg);//publish to stop the robot
        break_loop.data = true;
        break;
    }
    else
    {
        check_moving_obj.data = false;
        obstacle_found.data = false;
    }
}


```c
else
{
    continue;
}
}

count++; 
}

pub.publish(msg_decission);
    sleep(1);//use sleep if you want to delay loop.
}

int main(int argc, char** argv)
{
    ros::init(argc, argv,"listener_kinect_camera_depth_points");
    ros::NodeHandle nh;
    nh.param<double>("scale_linear", scale_linear_, 1.0);
    nh.param<double>("scale_angular", scale_angular_, 1.0);
    ros::Subscriber sub = nh.subscribe<PointCloud>("/camera/depth/points", 1, callback);

    pub = nh.advertise<std_msgs::Bool> ("/kinect_obstacle_detect", 1);
    dec_pub = nh.advertise<geometry_msgs::Twist>("/cmd_vel", 1, true);
    ros::spin();
}
```

7.3 Moving Obstacle detection by Kinect Sensors

The pseudo code of the algorithm for moving obstacle detection is given in figure 40 and discussed in section 4.4.3. The implementation code is given below.

```c
#include <ros/rosh>
#include <std_msgs/Bool.h>
#include "std_msgs/String.h"
#include "geometry_msgs/Twist.h"
#include <sensor_msgs/LaserScan.h>
#include <pcl_ros/point_cloud.h>
#include <pcl/point_types.h>
#include <boost/foreach.hpp>
#include <pcl/point_types.h>
#include <pcl/filters/passthrough.h>
```
#include <stdio.h>
#include <image_transport/image_transport.h>
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>

namespace enc = sensor_msgs::image_encodings;

using namespace std;
using namespace cv;

sensor_msgs::Image img_primary, img_previous, img_current;
std_msgs::Bool check_moving_obj, obstacle_found, is_moving_obs;
bool check_moving_obs = true;

ros::Publisher pub, dec_pub, mov_obs_pub;
std::string WINDOW_NAME = "current Image";
geometry_msgs::Twist cmd_vel_msg_;
double scale_linear_, scale_angular_;
double vel_x, vel_y, vel_omega;

ofstream myfile;

void dec_callback(const std_msgs::Bool::ConstPtr& msg)
{
    check_moving_obs = msg->data;
}

void callback(const sensor_msgs::Image::ConstPtr& msg)
{
    cv_bridge::CvImagePtr cv_ptr_pri, cv_ptr_prev, cv_ptr_cur;
    cv::Mat mat_pri, mat_pri_i, mat_cur, mat_cur_i, mat_pre, mat_pre_i, differenceImage,
    differenceImage_1, differenceImage_2, thresholdImage, finalImage;
    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;
    RNG rng(12345);

    if(img_previous.data.size() == 0)
    {
        img_previous = *msg;
        img_primary = *msg;
    }
else
{
    img_primary = img_previous;
    img_previous = img_current;
}

img_current = *msg;
if(check_moving_obs)
{
    try
    {
        cv_ptr_pri = cv_bridge::toCvCopy(img_primary,enc::TYPE_8UC1);//cv_bridge::toCvCopy(img_current, enc::BGR8);
        cv_ptr_prev = cv_bridge::toCvCopy(img_previous, enc::TYPE_8UC1);//TYPE_16UC1
        cv_ptr_cur = cv_bridge::toCvCopy(msg,enc::TYPE_8UC1);//cv_bridge::toCvCopy(img_current, enc::BGR8);

        mat_pri = cv_ptr_pri.image;
        cv::GaussianBlur( mat_pri, mat_pri, cv::Size(3,3), 0, 0, cv::BORDER_DEFAULT );
        cv::medianBlur( mat_pri, mat_pri, 3 );

        mat_cur = cv_ptr_cur->image;
        cv::GaussianBlur( mat_cur, mat_cur, cv::Size(3,3), 0, 0, cv::BORDER_DEFAULT );
        cv::medianBlur( mat_cur, mat_cur, 3 );

        mat_pre = cv_ptr_prev->image;
        cv::GaussianBlur( mat_pre, mat_pre, cv::Size(3,3), 0, 0, cv::BORDER_DEFAULT );
        cv::medianBlur( mat_pre, mat_pre, 3 );

        int count_diff=0;
        bool motion_detected = false;
        cv::absdiff(mat_cur,mat_pre,differenceImage_1);
        cv::absdiff(mat_pre,mat_pri,differenceImage_2);
        cv::bitwise_and(differenceImage_1, differenceImage_2, differenceImage);
        //threshold intensity image at a given sensitivity value
        cv::threshold(differenceImage,thresholdImage, 10,255,cv::THRESH_BINARY);

        //morphological operation //+dilation -erosion
        cv::dilate(thresholdImage, finalImage, Mat(), Point(-1, -1), 2, 1, 1);

        // Find contours
        findContours( finalImage, contours, hierarchy, CV_RETR_TREE,
                      CV_CHAIN_APPROX_SIMPLE, Point(0, 0) );
        // Draw contours
        Mat drawing = Mat::zeros( finalImage.size(), CV_8UC3 );
        cout<<"contour size : "<<contours.size()<<"\n";
double area=0, areamax=0;

int maxn=0;
std::string result;
std::stringstream sstm;

for( int i = 0; i < contours.size(); i++ )
{
    area=contourArea(contours[i], false);
    sstm << area << "\n";

    if(area>areamax)
    {
        areamax=area;
        //maxitem=i;
        maxn=i;
    }
    if(area > 1000)
    {
        is_moving_obs.data = true;
        Scalar color = Scalar( rng.uniform(0, 255), rng.uniform(0,255), rng.uniform(0,255) );
        try{
            drawContours( drawing, contours, i, color, 2, 8, hierarchy, 0, Point() );
        }
        catch (cv_bridge::Exception& e)
        {
            //if there is an error during conversion, display it
            ROS_ERROR("draw contour problem cv_bridge exception: %s", e.what());
            return;
        }
        vel_x = 0.0;
        vel_y = 0.0;
        vel_omega = 0.0;
        cout<<"something moving... :"<<"\n";
        cmd_vel_msg_.linear.x = scale_linear_* vel_x;
        cmd_vel_msg_.linear.y = scale_linear_* vel_y;
        cmd_vel_msg_.angular.z = scale_angular_* vel_omega;
        dec_pub.publish(cmd_vel_msg_);
        break;
    }
    else
    {
        is_moving_obs.data = false;
        vel_x = 0.0;
        vel_y = 0.0;
    }
vel_omega = 0.0;

     cmd_vel_msg_.linear.x = scale_linear_ * vel_x;
     cmd_vel_msg_.linear.y = scale_linear_ * vel_y;
     cmd_vel_msg_.angular.z = scale_angular_ * vel_omega;
     dec_pub.publish(cmd_vel_msg_);
     }
     }

myfile << sstm.str();

// Show in a window
//show the difference image and threshold image

cv::namedWindow(WINDOW_NAME);
     cv::imshow(WINDOW_NAME,cv_ptr_cur->image);
     cv::waitKey(1);
     cv::namedWindow("pre image");
     cv::imshow("pre image",cv_ptr_prev->image);
     cv::waitKey(1);
     cv::namedWindow("Difference Image");
     cv::imshow("Difference Image",differenceImage);
     cv::waitKey(1);
     cv::namedWindow("Threshold Image");
     cv::imshow("Threshold Image", thresholdImage);
     cv::waitKey(1);
     cv::namedWindow("Final Image");
     cv::imshow("Final Image", finalImage);
     cv::waitKey(1);
     cv::namedWindow("Contours", CV_WINDOW_AUTOSIZE );
     cv::imshow("Contours", drawing );
     cv::waitKey(1);

if(motion_detected)
{
    cout<<"------------------------ Motion Detected --------""<<"n";
}

}

catch (cv_bridge::Exception& e)
{
    //If there is an error during conversion, display it
    ROS_ERROR("cv_bridge exception: %s", e.what());
}
int main(int argc, char** argv)
{
    try{
        myfile.open("example.csv");
    } catch(...) {}
    ros::init(argc, argv,"listener_depth_comparer");
    ros::NodeHandle nh;
    nh.param<double>("scale_linear", scale_linear_, 1.0);
    nh.param<double>("scale_angular", scale_angular_, 1.0);
    ros::Subscriber sub = nh.subscribe<sensor_msgs::Image>("/camera/depth/image_raw", 1, callback);
    ros::Subscriber dec_sub = nh.subscribe<std_msgs::Bool>("/kinect_obstacle_detect", 1, dec_callback);
    dec_pub = nh.advertise<geometry_msgs::Twist>("/cmd_vel", 1, true);
    mov_obs_pub = nh.advertise<std_msgs::Bool>("/moving_detection", 1);
    cv::destroyWindow(WINDOW_NAME);
    ros::spin();
}

Bibliography


[38] Gehrig, S. K., & Stein, F. J. (2007). Collision avoidance for vehicle-following systems. IEEE transactions on intelligent transportation systems, 8(2), 233-244.


