

OBSTACLE AVOIDANCE BASED ON MACHINE LEARNING

Submitted in partial fulfilment of the requirements
of the degree of

Bachelor of Engineering

By

Shrey Gupta	BE – 7, 13
Aarushi Nandoskar	BE – 7, 31
Narendra Parihar	BE – 7, 34
Surya Prakash Yadav	BE – 7, 60

Under the Guidance of

Internal Guide:

Dr. T. P. Vinutha

Internal Co - Guide:

Prof. Amisha Bhoir

External Guide:

Shri. Gaurav Baluni



**DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATION
ENGINEERING**

**SHAH AND ANCHOR KUTCHHI ENGINEERING
COLLEGE**

**MUMBAI UNIVERSITY
2022 – 2023**



Mahavir Education Trust's

SHAH & ANCHOR KUTCHHI ENGINEERING COLLEGE

Mahavir Education Trust Chowk, W.T. Patil Marg, Chembur, Mumbai 400 088.

Affiliated to University of Mumbai, Approved by D.T.E. & A.I.C.T.E.



ISO 9001:2015 Certified

Certificate

This is to certify that the report of the project entitled

OBSTACLE AVOIDANCE BASED ON MACHINE LEARNING

is a bonafide work of

Shrey Gupta	BE – 7, 13
Aarushi Nandoskar	BE – 7, 31
Narendra Parihar	BE – 7, 34
Surya Prakash Yadav	BE – 7, 60

submitted to the

UNIVERSITY OF MUMBAI

*during semester VII in partial fulfilment of the requirement for the award of
the degree of*

BACHELOR OF ENGINEERING

in

ELECTRONICS AND TELECOMMUNICATION ENGINEERING

Dr. T. P. Vinutha
Internal Guide

Prof. Amisha Bhoir
Internal Co-Guide

Dr. T. P. Vinutha
I/c Head of Department

Dr. Bhavesh Patel
Principal



Bhabha Atomic Research Centre
Security Electronics & Cyber Technology Division

Gaurav Baluni
Scientific Officer 'F'
Email: gbaluni@barc.gov.in

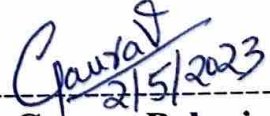
BARC Trombay
Mumbai - 400085
Tel: +91-22-25590165

CERTIFICATE

This is to certify that the following students,

1. Shrey Gupta
2. Aarushi Nandoskar
3. Narendra Parihar
4. Surya Prakash Yadav

of Electronics and Telecommunication Engineering department of Shah and Anchor Kutchhi Engineering College, Chembur have carried out project work entitled '**Obstacle Avoidance based on Machine Learning**' in this organization from 14th September, 2022 to 26th April, 2023.



Shri. Gaurav Baluni
External Guide

वैज्ञानिक अधिकारी / Scientific Officer
सुरक्षा इलेक्ट्रॉनिक्स एवं साइबर प्रौद्योगिकी प्रभाग
Security Electronics and Cyber Technology Division
भाषा परमाणु अनुसंधान केंद्र / Bhabha Atomic Research Centre
भारत सरकार / Government of India
ट्रॉम्बे, मुंबई - 400085, भारत / Trombay, Mumbai - 400 085, India

To,
The Principal
Shah and Anchor Kutchhi Engineering College,
Chembur, Mumbai - 88.

Subject: Confirmation of Attendance

Respected Sir,

This is to certify that the final year (BE) students from your college,

1. Shrey Gupta
2. Aarushi Nandoskar
3. Narendra Parihar
4. Surya Prakash Yadav

have duly attended the sessions on the day allotted to them during the period from 14th September, 2022 to 26th April, 2023 for performing the Project titled, '**Obstacle Avoidance based on Machine Learning**'. They were punctual and regular in their attendance. Following is the detailed record of the student's attendance.

Attendance Record:

Date	Shrey Gupta	Aarushi Nandoskar	Narendra Parihar	Surya Prakash Yadav
14 - 9 - 2022	Present	Present	Present	Present
21 - 9 - 2022	Present	Present	Present	Present
28 - 9 - 2022	Present	Present	Present	Present
12 - 10 - 2022	Present	Present	Present	Present
9 - 11 - 2022	Present	Present	Present	Present
16 - 11 - 2022	Present	Present	Present	Present
23 - 11 - 2022	Present	Present	Present	Present
21 - 12 - 2022	Present	Present	Present	Present
11 - 1 - 2023	Present	Present	Absent	Absent
18 - 1 - 2023	Present	Present	Present	Present
25 - 1 - 2023	Present	Present	Present	Present
1 - 2 - 2023	Present	Present	Present	Present
8 - 2 - 2023	Present	Absent	Absent	Present
15 - 2 - 2023	Present	Present	Present	Present

1 - 3 - 2023	Present	Present	Present	Present
8 - 3 - 2023	Present	Absent	Present	Present
15 - 3 - 2023	Present	Present	Present	Present
29 - 3 - 2023	Present	Present	Absent	Present
5 - 4 - 2023	Present	Present	Present	Absent
12 - 4 - 2023	Present	Absent	Present	Absent
19 - 4 - 2023	Present	Present	Present	Present
26 - 4 - 2023	Present	Present	Absent	Present

PROJECT REPORT APPROVAL FOR B.E.

This project report entitled "Obstacle Avoidance based on Machine Learning" by ***Shrey Gupta, Aarushi Nandoskar, Narendra Parihar and Surya Prakash Yadav*** is approved for the degree of Bachelor of Engineering.

Name and Signature of the Examiner

1. -----

2. -----

Date:

Place:

DECLARATION

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

(Signature)

Shrey Gupta

(Signature)

Aarushi Nandoskar

(Signature)

Narendra Parihar

(Signature)

Surya Prakash Yadav

Date:

ABSTRACT

Obstacle avoidance is a fundamental requirement for autonomous robots which operate in, and interact with, the real world. Vision-based obstacle avoidance is challenging due to the difficulties of estimating obstacle depth from 2D images. When perception is limited to monocular vision, avoiding collision becomes significantly more challenging due to the lack of 3D information. In this project, we have used the SIFT algorithm for goal detection and combined it with monocular depth estimation to enable an autonomous vehicle to reach its goal while avoiding obstacles. U-Net architecture for monocular depth estimation has been used to provide depth information to an autonomous vehicle, allowing it to detect obstacles and avoid collisions. To train the U-Net model for monocular depth estimation, NYU Depth V2 dataset is used which contains RGB-D images of indoor scenes. Segmenting a depth image into left, right, and center portions and using each portion to check for obstacles is a technique that has been used for obstacle avoidance. It is observed that the accuracy of monocular depth estimation using U-Net is affected by various factors, such as lighting conditions, object sizes, and distances.

INDEX

Sr. No.	Name of Topic	Page Number
1.	INTRODUCTION.....	1
1.1	Objective.....	1
1.2	Problem Definition.....	2
1.3	Methodology.....	2
1.4	Organization of the Report.....	2
2.	LITERATURE REVIEW.....	4
2.1	Autonomous Robots.....	4
2.2	Obstacle Detection and Avoidance.....	4
2.3	Bug Algorithms.....	5
2.4	Monocular Depth Estimation.....	6
3.	WHEELED MOBILE ROBOTS.....	7
3.1	Wheel Design and Geometry.....	7
3.2	Wheel Kinematic Constraints.....	7
3.3	Mobile Robot Kinematics.....	8
4.	ROBOT NAVIGATION.....	10
4.1	Sensors.....	11
5.	OBSTACLE AVOIDANCE ALGORITHMS.....	13
5.1	Bug 0 Algorithm.....	13
5.2	Bug 1 Algorithm.....	14
5.3	Bug 2 Algorithm.....	14
6.	MONOCULAR DEPTH ESTIMATION.....	16
6.1	NYU Depth V2 Dataset.....	17
6.2	U-Net Architecture.....	18

7.	IMPLEMENTATION.....	19
7.1	Monocular Depth Estimation.....	19
7.2	Obstacle Avoidance.....	22
7.3	Goal Detection.....	24
8.	HARDWARE AND SOFTWARE.....	25
8.1	TurtleBot.....	25
8.2	ROS.....	26
8.3	Gazebo.....	26
9.	PROGRAMMING.....	27
9.1	Obstacle Avoidance.....	27
9.2	Goal Detection with Obstacle Avoidance.....	32
10.	RESULTS AND DISCUSSION.....	47
10.1	Error Analysis.....	52
11.	TROUBLESHOOTING.....	56
11.1	Problems Faced.....	56
11.2	Solution to the Problems.....	57
12.	APPLICATIONS.....	59
13.	CONCLUSION.....	60
14.	FUTURE PLAN.....	61
	REFERENCES.....	62
	ACKNOWLEDGEMENT.....	63
	ANNEXURES.....	64

LIST OF FIGURES

1.	Fig. 1.1 Illustration of Obstacle Avoidance.....	1
2.	Fig. 3.3 A Differential-drive Robot in its Global reference frame.....	8
3.	Fig. 4 Flowchart for Robot navigation.....	10
4.	Fig. 5.1 Illustration of Bug 0 Algorithm.....	13
5.	Fig. 5.2 Illustration of Bug 1 Algorithm.....	14
6.	Fig. 5.3 Illustration of Bug 2 Algorithm.....	15
7.	Fig. 6.1 NYU Depth V2 Labeled Dataset.....	17
8.	Fig. 6.2 U-Net Architecture.....	18
9.	Fig. 7.1 Flowchart for Monocular Depth Estimation.....	21
10.	Fig. 7.2 Flowchart for Obstacle Avoidance.....	23
11.	Fig. 8.1 TurtleBot 2.....	25
12.	Fig. 8.3 Gazebo.....	26
13.	Fig. 10.1 Input RGB images and its predicted Depth maps.....	47
14.	Fig. 10.2 Accuracy and Loss graph of Training vs Validation.....	48
15.	Fig 10.3 Performance metrics of the Model.....	48
16.	Fig. 10.4 Robot in an Environment with obstacles.....	51
17.	Fig. 10.5 Estimated Goal Position.....	51
18.	Fig. 10.6 Depth predictions for Ground object.....	53
19.	Fig. 10.7 Depth predictions for Elevated object.....	54

LIST OF TABLES

1.	Table 10.1 Performance analysis of the Model.....	50
2.	Table 10.2 Error analysis for Ground object.....	53
3.	Table 10.3 Error analysis for Elevated object.....	54

1. INTRODUCTION

1.1 OBJECTIVE

In recent years, the mobile robot has received considerable attention from researchers for its application in various environments. For a mobile robot navigating its way from starting point to a goal point while traversing through deterrents, it needs to recognize the obstacles and generate new trajectories to reach the destination. Obstacle avoidance is a task in local path planning and this ensures the safety of humans and robots. To avoid collisions during motion, the mobile robot must detect boundaries of obstacles, create new trajectories, and calculate instant velocity and head angular. It is essential because it ensures the safe and efficient navigation of the robot through its environment. There are numerous methods to implement obstacle avoidance. One of the most common methods is the use of sensors, such as LiDARs and cameras, which can detect obstacles and can be used to plan a safe trajectory for the robot to follow.

In addition to obstacle avoidance, other factors must be considered for safe robot navigation. One such factor is the safe distance from obstacles. The robot must maintain a safe distance from obstacles to avoid collisions. Another factor is path smoothness, which refers to the smoothness of the robot's trajectory. Smooth trajectories are essential for stable and efficient robot motion. Cost is another factor that must be considered in robot navigation. Cost refers to the energy and time required for the robot to reach its destination. A good navigation strategy should minimize cost while ensuring safe and efficient robot motion. Vision-based sensors like camera sensors are attractive for obstacle avoidance because they provide rich information about the environment at a low cost. Compared to other sensors such as LiDAR or sonar, vision-based sensors offer higher resolution, greater range, and can provide depth information. Furthermore, monocular cameras are more cost-effective than stereo cameras and can still provide depth information using monocular depth estimation techniques. The use of a single camera simplifies the robot's hardware and reduces cost, while still providing adequate information for obstacle avoidance and goal reaching.

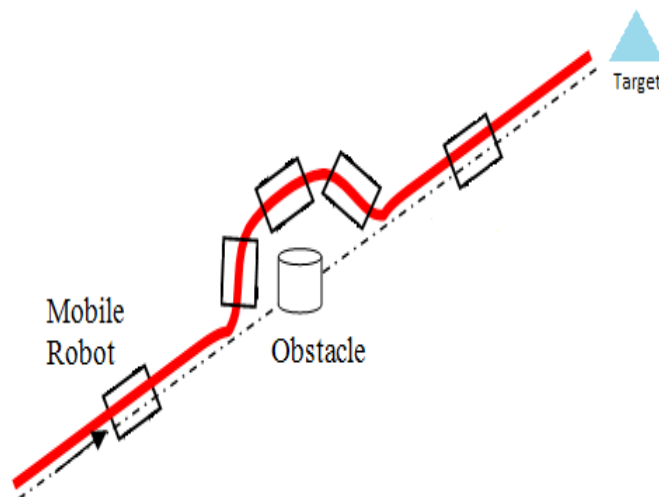


Fig 1.1 Illustration of Obstacle Avoidance

1.2 PROBLEM DEFINITION

Obstacle avoidance is a primary requirement of any autonomous robot. Any robot that is to drive autonomously must be able to detect and avoid obstacles that it might encounter. The main objective of this project is to build an autonomous robot by incorporating obstacle avoidance and goal reaching capabilities that allows the robot to safely and efficiently navigate in unknown environment without human intervention by avoiding collisions.

1.3 METHODOLOGY

When mobile robots operate in the real world, subject to ever varying conditions, one of the fundamental capabilities they need is to be able to avoid obstacles. Local obstacle avoidance focuses on changing the robot's trajectory as informed by its sensors during robot motion. The resulting robot motion is both a function of the robot's current or recent sensor readings and its goal position and relative location to the goal position. For obstacle avoidance, Monocular depth estimation is a critical component of the system. It is a computer vision task that involves estimating the depth or distance of objects in a scene from a single image or video frame captured by a monocular camera. There are several techniques for monocular depth estimation, including traditional methods and deep learning-based methods. To avoid collisions, the depth image is segmented into left, right and center portions, and each portion is used to check for obstacles. In addition to obstacle avoidance, the robot also needs to locate and reach a goal in the environment. To achieve this, Scale Invariant Feature Transform (SIFT) algorithm can be used to detect distinctive features in the robot's camera image that correspond to the goal. By matching these features to those in a reference image of the goal, the robot can determine its location and orientation relative to the goal. By using a single camera and incorporating SIFT algorithm, the system can estimate the depth of the scene and detect obstacles and goals in real-time, making it a cost-effective solution for obstacle avoidance and goal reaching tasks. We have implemented the obstacle avoidance and goal reaching framework in Robotic Operating System (ROS) using Python programming language.

1.4 ORGANIZATION OF THE REPORT

The report has been organized into multiple chapters to provide a comprehensive and structured presentation of the project.

Chapter 1 serves as the introduction, outlining the objectives, problem definition and methodology used.

Chapter 2 is dedicated to the literature review and provides an in-depth analysis of the existing research.

Chapter 3 provides an in-depth discussion of wheel mobile robots.

Chapter 4 delves into robot navigation.

Chapter 5 focuses on the obstacle avoidance algorithms.

Chapter 6 provides a comprehensive overview of monocular depth estimation.

Chapter 7 elaborates on the implementation of monocular depth estimation, obstacle avoidance and goal detection.

Chapter 8 covers the programming aspects of the project.

Chapter 9 describes the hardware and software used for implementation.

Chapter 10 discusses the results.

Chapter 11 highlights the problems encountered and their corresponding solutions.

Chapter 12 explores the application of mobile robots in obstacle avoidance.

Chapter 13 summarizes the findings in the conclusion.

Chapter 14 provides insights into the future plan for this project.

2. LITERATURE REVIEW

The present study aims to conduct a comprehensive review of relevant literature pertaining to autonomous robots. The focus of this literature review will be on topics such as obstacle detection and avoidance, bug algorithms, and monocular depth estimation. The aim is to provide an in-depth analysis of the current state of knowledge in these areas, including key findings and gaps in the existing research.

2.1 AUTONOMOUS ROBOTS

The neural network algorithm is developed and implemented in [1] for mobile robot and programmed in the C language. Neural networks are well suited to mobile robots because they can operate with imprecise information. This paper [2] provides a comprehensive survey of key studies in the implementation of AI in Avs. It analyzes the use of AI in supporting the primary applications in AVs: 1. Perception, 2. Localization and Mapping and 3. Decision making. It provides insights into potential opportunities regarding the use of AI in conjunction with other emerging technologies. Despite a substantial amount of research efforts, there still exists challenging problems in using AI for supporting AVs' perception, localization and mapping, and decision making. Many AI approaches such as ML, DL, and RL solutions have been applied to help AVs better sense surroundings and make human-like decisions. However, such applications have been inherently limited by data availability and data quality, complexity and uncertainty, complex model tuning, and hardware restrictions.

2.2 OBSTACLE DETECTION AND AVOIDANCE

By using R-CNN (Region-based Convolutional Neural Network) on a micro controller like Arduino, the algorithm presented in [3] learns to control the robot in an unknown environment. This method has potential to improve the performance of the robot to avoid obstacles in a highly occupied environment and problematic scenario. A bidirectional feature pyramid networks combining hybrid network architecture of ResNet and improved DenseNet, is proposed in [4] to solve the problem of low detection and classification accuracy of obstacles in the visual inspection of USV. The results show that the proposed method has the highest performance for obstacle detection and is more suitable for the application of USV (Unmanned Surface Vehicle). Use of Artificial Intelligence System (AIS) and some navigation methods based on Artificial Neural Networks (ANNs) to training datasets in [5] provided high Mean Square Error (MSE) from training on MATLAB Simulink tool. Standardization techniques are used

to improve the performance. The results show a high improvement in the efficiency with the shortest training time of the neural networks; as well as a very well trained and good generalization performance. In [6], development of a neural network robot which is divided into hardware and OS (Operating System) part is proposed. In hardware, this robot has embedded systems that include an AVR microcontroller. This robot has an OS written in C programming language. To train this robot a feed-forward back-propagation network is used. The robot working accuracy in Daylight environment is very low cause of other light disturbance. In the Semi dark and Dark environment, the robot operation accuracy is very high cause there is no other disturbance of light detected by the robot and having only one light source, the robot works very efficiently. [7] Proposes intelligent control of an autonomous robot which is trained with ANN to navigate in an environment which is full of static obstacles. Ultrasonic sonar sensors are used with the Neural Network in order to find its route without colliding with any obstacle after the training. The robot successfully navigates a path which is full of static obstacles. The Neural Network is trained in a supervised way with the back-propagation algorithm. Different autonomous industry and systems would find this work applicable and relevant to their interests. In [8], development and implementation of neural control systems in mobile robots in obstacle avoidance in real time using ultrasonic sensors with complex strategies of decision-making in development (MATLAB and Processing) is presented. An Arduino embedded platform is used to implement the neural control for field results. Neural networks are excellent tools applicable in mobile robots evading obstacles, which have the ability to work with imprecise information. [9] Proposes applying Neural Network Algorithm to learn from the sonar sensory data in order to navigate the mobile robot to move a collision free trajectory. Stability of the robot's movement depends on system sensitivity. There should be appropriate distance which is suitable for the maze in undefined environment.

2.3 BUG ALGORITHMS

A nonholonomic wheeled mobile robot with a forward-looking sensor is depicted in [10] where the detection range is limited. A bug-inspired algorithm has been proposed which is based on combination of path following method and Bug 2 algorithm. The mobility of the robot is not only limited by its nonholonomic constraints, but is also bounded by a certain velocity range. The robot is able to avoid collisions and reach the destinations with very small steady-state position errors.

2.4 MONOCULAR DEPTH ESTIMATION

Different kinds of network frameworks, loss functions and training strategies are proposed in [11] in order to improve the accuracy of depth estimation. Some representative existing methods according to different training manners: supervised, unsupervised and semi-supervised are reviewed. Also, the challenges and some ideas for future researches in monocular depth estimation are discussed. Studying the mechanism of depth estimation is a promising direction, which may effectively improve the accuracy, transferability and real-time performance. A convolutional neural network for computing a high-resolution depth map given a single RGB image with the help of transfer learning is presented in [12]. Following standard encoder-decoder architecture, features extracted are leveraged using high performing pre-trained networks when initializing the encoder along with augmentation and training strategies that lead to more accurate results. This method is able to achieve detailed high-resolution depth maps. The network, with fewer parameters and training iterations, outperforms state-of-the-art on two datasets and also produces qualitatively better results that capture object boundaries more faithfully.

3. WHEELED MOBILE ROBOTS

The wheel has been by far the most popular locomotion mechanism in mobile robotics and in man-made vehicles in general. It can achieve very good efficiencies and does so with a relatively simple mechanical implementation. In addition, balance is not usually a research problem in wheeled robot designs, because wheeled robots are almost always designed so that all wheels are in ground contact at all times. When more than three wheels are used, a suspension system is required to allow all wheels to maintain ground contact when the robot encounters uneven terrain. Instead of worrying about balance, wheeled robot research tends to focus on the problems of traction and stability, maneuverability, and control.

3.1 WHEEL DESIGN AND GEOMETRY

There are four major wheel classes. They differ widely in their kinematics, and therefore the choice of wheel type has a large effect on the overall kinematics of the mobile robot. The standard wheel and the castor wheel have a primary axis of rotation and are thus highly directional. The key difference between these two wheels is that the standard wheel can accomplish this steering motion with no side effects, as the center of rotation passes through the contact patch with the ground, whereas the castor wheel rotates around an offset axis, causing a force to be imparted to the robot chassis during steering. The Swedish wheel and the spherical wheel are both designs that are less constrained by directionality than the conventional standard wheel. The Swedish wheel functions as a normal wheel, but provides low resistance in another direction as well, sometimes perpendicular to the conventional direction, as in the Swedish 90, and sometimes at an intermediate angle, as in the Swedish 45. The key advantage of this design is that, although the wheel rotation is powered only along the one principal axis through the axle, the wheel can kinematically move with very little friction along many possible trajectories, not just forward and backward. The spherical wheel is a truly omnidirectional wheel, often designed so that it may be actively powered to spin along any direction. The choice of wheel types for a mobile robot is strongly linked to the choice of wheel arrangement, or wheel geometry. The mobile robot designer must consider these two issues simultaneously when designing the locomoting mechanism of a wheeled robot.

3.2 WHEEL KINEMATIC CONSTRAINTS

The first step to a kinematic model of the robot is to express constraints on the motions of individual wheels. The motions of individual wheels can later be combined to compute the motion of the robot as a whole. There are four basic wheel types with widely varying kinematic properties. Therefore, we begin

by presenting sets of constraints specific to each wheel type. We assume that the plane of the wheel always remains vertical and that there is in all cases one single point of contact between the wheel and the ground plane. Furthermore, we assume that there is no sliding at this single point of contact. That is, the wheel undergoes motion only under conditions of pure rolling and rotation about the vertical axis through the contact point. Under these assumptions, we present two constraints for every wheel type. The first constraint enforces the concept of rolling contact that is the wheel must roll when motion takes place in the appropriate direction. The second constraint enforces the concept of no lateral slippage that is the wheel must not slide orthogonal to the wheel plane.

3.3 MOBILE ROBOT KINEMATICS

In mobile robotics, we need to understand the mechanical behaviour of the robot both in order to design appropriate mobile robots for tasks and to understand how to create control software for an instance of mobile robot hardware. Deriving a model for the whole robot's motion is a bottom-up process. Each individual wheel contributes to the robot's motion and, at the same time, imposes constraints on robot motion. Wheels are tied together based on robot chassis geometry, and therefore their constraints combine to form constraints on the overall motion of the robot chassis.

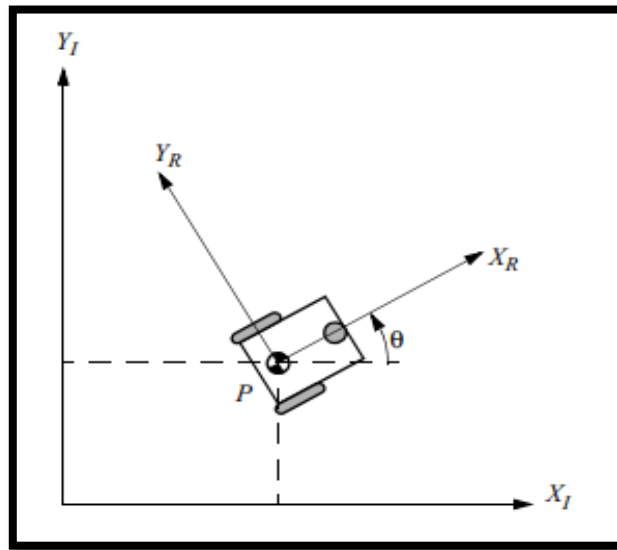


Fig 3.3 A Differential-drive Robot in its Global reference frame

Consider that a differential drive robot has two wheels, each with diameter r . Given a point P centered between the two drive wheels, each wheel is a distance l from P . Given r , l , θ , and the spinning speed of each wheel φ_1 and φ_2 , a forward kinematic model would predict the robot's overall speed in the global reference frame:

$$\xi_I = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = f(r, l, \theta, \varphi_1, \varphi_2)$$

Suppose that the robot's local reference frame is aligned such that the robot moves forward along $+X_R$, as shown in the figure 3.3. First consider the contribution of each wheel's spinning speed to the translation speed at P in the direction of $+X_R$. If one wheel spins while the other wheel contributes nothing and is stationary, since P is halfway between the two wheels, it will move instantaneously with half the speed: $x_{r1} = (1/2)r\varphi_1$ and $x_{r2} = (1/2)r\varphi_2$. In a differential drive robot, these two contributions can simply be added to calculate the x_R component of ξ_R .

Consider, for example, a differential robot in which each wheel spins with equal speed but in opposite directions. The result is a stationary, spinning robot. As expected, x_R will be zero in this case. The value y_R of is even simpler to calculate. Neither wheel can contribute to sideways motion in the robot's reference frame and so y_R is always zero. Finally, we must compute the rotational component of θ_R . Once again, the contributions of each wheel can be computed independently and just added.

Consider the right wheel (wheel 1). Forward spin of this wheel results in counter clockwise rotation at point P. If wheel 1 spins alone, the robot pivots around wheel 2. The rotation velocity ω_1 at P can be computed because the wheel is instantaneously moving along the arc of a circle of radius $2l$:

$$\omega_1 = \frac{r\varphi_1}{2l}$$

The same calculation applies to the left wheel, with the exception that forward spin results in clockwise rotation at point P:

$$\omega_2 = \frac{r\varphi_2}{2l}$$

Combining these individual formulas yields a kinematic model for the differential-drive robot:

$$\xi_I = R(\theta)^{-1} \begin{bmatrix} \frac{r\varphi_1}{2} + \frac{r\varphi_2}{2} \\ 0 \\ \frac{r\varphi_1}{2l} + \frac{-r\varphi_2}{2l} \end{bmatrix}$$

where,

$$R(\theta)^{-1} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

4. ROBOT NAVIGATION

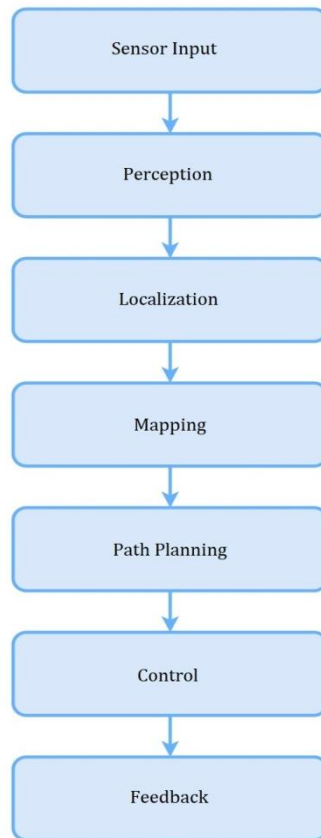


Fig 4 Flowchart for Robot navigation

- **Sensor Input:**

This component includes various sensors that the robot uses to gather data about its surroundings. For example, cameras can provide visual data, LIDAR can provide depth data, sonar can provide distance data, and infrared can provide temperature data. The specific sensors used will depend on the robot's design and application.

- **Perception:**

Once the sensor input is collected, the robot's onboard computer processes the data to create a representation of the environment. This can involve techniques such as image processing, point cloud analysis, and machine learning to identify objects, obstacles, and other relevant features. The resulting representation is used to inform the robot's navigation decisions.

- **Localization:**

Using the data from perception, the robot must determine its own position and orientation within the environment. This can be achieved through techniques such as Simultaneous Localization and Mapping (SLAM), which combines sensor data with a map of the environment to estimate the robot's position.

- **Mapping:**

The robot creates a map of the environment based on the perception and localization data it has gathered.

This map can be used to plan efficient and safe paths for the robot to navigate through the environment.

- **Path Planning:**

Given the map of the environment, the robot must determine the most efficient and safe path to navigate through it. This can involve algorithms such as A* or Dijkstra's algorithm to find the shortest path while avoiding obstacles and other constraints.

- **Control:**

Once the path is planned, the robot's control system must execute the plan by sending signals to the motors or other actuators that control its movement. This involves adjusting the robot's speed and direction to follow the planned path.

- **Feedback:**

As the robot navigates through the environment, it must continually monitor its sensors and adjust its path as needed to account for changes in the environment or unexpected obstacles. This feedback loop ensures that the robot is able to navigate effectively and safely.

4.1 SENSORS

An autonomous vehicle consists of several major sensors. Each type of sensor presents advantages and drawbacks in autonomous vehicles.

- **Cameras:**

Cameras are widely used in robot navigation for various tasks, such as object recognition and tracking. Object recognition involves detecting and identifying objects in the environment, which can include people, vehicles, or other obstacles. This information is then used by the robot's control system to adjust its path and avoid collisions. Object tracking involves following the movement of an object over time, which can be used for tasks such as tracking a moving vehicle or pedestrian.

- **GPS/IMU:**

The GPS/IMU system helps the autonomous vehicle localize itself by reporting both inertial updates and a global position estimate at a high rate. GPS is a fairly accurate localization sensor, but its update rate is slow, and thus not capable of providing real-time updates. However, IMU errors accumulate over time, leading to a corresponding degradation in the position estimates. Nonetheless, an IMU can provide updates more frequently. This should satisfy the real-time requirement. By combining both GPS and IMU, we can provide accurate and real-time updates for vehicle localization.

- **LiDAR:**

LiDAR is used for mapping, localization, and obstacle avoidance. It works by bouncing a beam off surfaces and measures the reflection time to determine distance. Due to its high accuracy, LiDAR can be used to produce HD maps, to localize a moving vehicle against HD maps, to detect obstacle ahead, etc.

- Radar and Sonar:

Radar works by emitting radio waves and detecting the reflections of those waves off of objects. By measuring the time, it takes for the waves to return and analyzing their properties, radar systems can determine the distance, direction, speed, and size of far objects. Sonar works in a similar way, but instead of using radio waves, it emits sound waves and detects the echoes produced by objects. The time it takes for the sound waves to bounce back and the intensity of the echoes can be used to calculate the distance, direction, and size of near objects.

Each type of sensor used in robot navigation has its own unique strengths and weaknesses. For example, GPS/IMU is a fairly accurate localization sensor, but its update rate is slow, while LiDAR is highly accurate and can produce HD maps, but it is expensive. Cameras are relatively inexpensive and can be used for object recognition and tracking tasks, but they are affected by lighting conditions and may not be as accurate as other sensors.

We chose vision sensor for robot navigation. It is because they provide rich and detailed information about the environment. Cameras can capture color, texture, and shape information that can be used for depth estimation and object tracking tasks. They are also relatively inexpensive compared to LiDAR and other sensors.

Obstacle avoidance is a critical component of robot navigation, as it allows the robot to safely navigate through its environment without colliding with obstacles. Obstacle avoidance is particularly important in dynamic environments, where the position and movement of obstacles may change over time. By continually monitoring its surroundings and adjusting its path as needed, a robot can safely navigate through its environment and accomplish its task.

5. OBSTACLE AVOIDANCE ALGORITHMS

Obstacle avoidance algorithms are designed to help mobile robots navigate their way around obstacles in order to reach their destination safely and efficiently. These algorithms can be classified into two main categories: reactive and deliberative. Reactive algorithms are designed to react to the current state of the environment, while deliberative algorithms are designed to plan a path around obstacles before the robot starts moving. One type of reactive algorithm is the bug algorithm.

Bug algorithms are a family of algorithms that use the idea of following the boundary of an obstacle to avoid it. There are several variations of the bug algorithm.

5.1 BUG 0 ALGORITHM

- Head towards goal
- Follow obstacles until you can head towards the goal again
- Continue

The bug 0 algorithm involves the robot following the boundary of the obstacle until it reaches a point where it can continue on its path towards the goal. The algorithm works by calculating the distance between the robot and the obstacle, and then turning the robot to follow the obstacle boundary. The robot then moves along the obstacle boundary until it reaches the point where it can continue towards the goal.

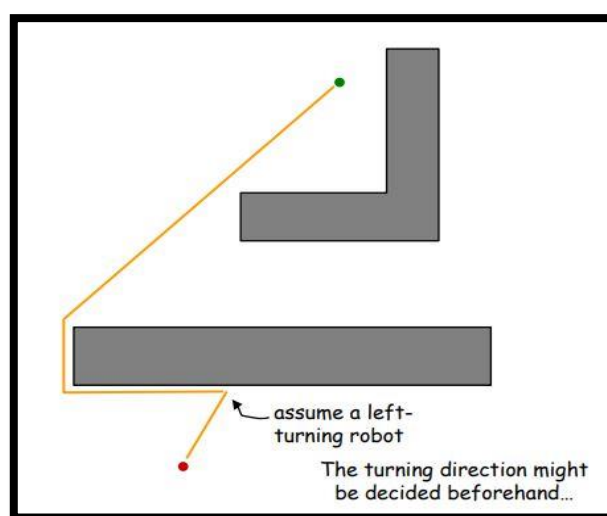


Fig 5.1 Illustration of Bug 0 Algorithm

5.2 BUG 1 ALGORITHM

- Head towards goal
- If an obstacle is encountered, circumnavigate it and remember how close you get to the goal
- Return to that closest point (by wall-following) and continue

The bug 1 algorithm works by having the robot move towards the goal until it encounters an obstacle. When an obstacle is detected, the robot moves towards the closest point on the obstacle's boundary. The robot then moves along the obstacle boundary until it reaches a point where it can continue on its path towards the goal.

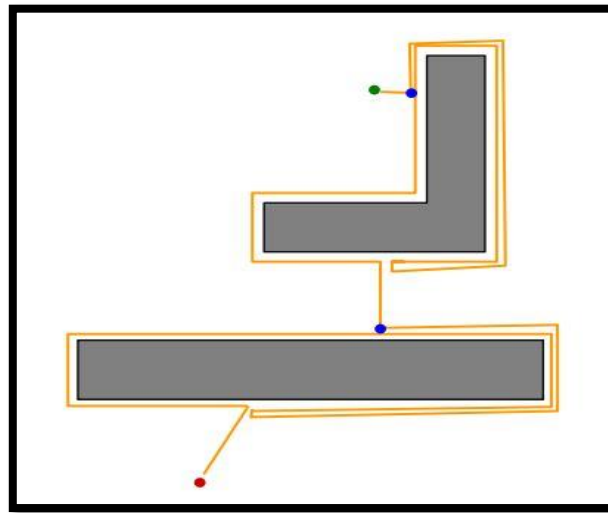


Fig 5.2 Illustration of Bug 1 Algorithm

5.3 BUG 2 ALGORITHM

- Head towards goal on the m-line
- If an obstacle is in the way, follow it until you encounter the m-line again closer to the goal
- Leave the obstacle and continue towards the goal

The bug 2 algorithm works by having the robot move towards the goal until it encounters an obstacle. The robot then moves along the obstacle boundary until it reaches the point where it can continue on its path towards the goal.

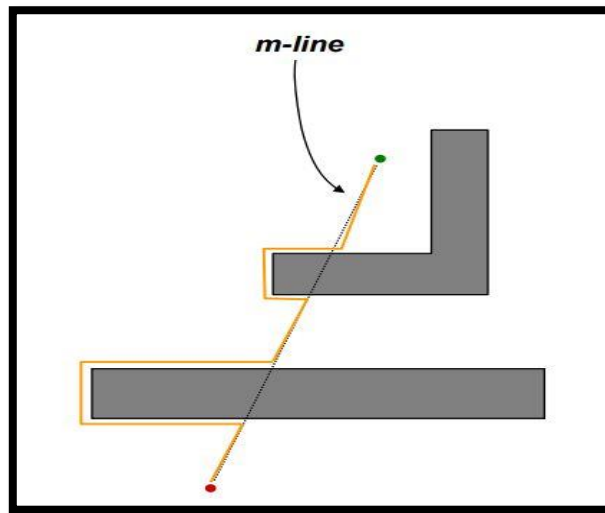


Fig 5.3 Illustration of Bug 2 Algorithm

Other obstacle avoidance techniques include the use of sensors such as LiDAR (Light Detection and Ranging), SONAR (Sound Navigation and Ranging), and Vision-based systems. LiDAR and SONAR are based on the principles of sending out a signal and measuring the time it takes for the signal to bounce back, which allows the robot to determine the distance to an obstacle. Vision-based systems, on the other hand, use cameras to capture images of the environment and use image processing techniques to detect obstacles.

Vision-based depth estimation can play a significant role in obstacle avoidance as it can provide rich information about the environment at a low cost. Monocular depth estimation is a technique that involves estimating the depth or distance of objects in a scene from a single image or video frame captured by a monocular camera. This technique can be used to segment the depth image into left, right and center portions, and each portion can be used to check for obstacles, making it a promising approach for obstacle avoidance in mobile robots.

6. MONOCULAR DEPTH ESTIMATION

Depth estimation is the process of calculating the distance between a camera and objects in a scene. This is an important task in many applications such as robotics, autonomous vehicles and augmented reality. There are various techniques that can be used for depth estimation, including stereo vision, time-of-flight cameras, structured light, and monocular depth estimation. Stereo vision involves using two cameras to capture an image of the same scene from slightly different angles. By comparing the differences between the two images, it is possible to calculate the depth of objects in the scene through triangulation. Triangulation is the process of using the known distance between the two cameras, the angle at which each camera views the object, and the position of the object in the two images to determine its position in three-dimensional space. Time-of-flight cameras use the time it takes for a light pulse to bounce off an object and return to the camera to estimate depth. Structured light involves projecting a pattern of light onto the scene and calculating depth from the distortion of the pattern. Monocular depth estimation is a newer approach that involves using a single camera and deep learning algorithms to estimate depth. This technique uses convolutional neural networks (CNNs) to learn a mapping from 2D images to 3D depth maps.

Monocular depth estimation is a computer vision task that involves estimating the depth or distance of objects in a scene from a single image or video frame captured by a monocular camera. There are several techniques for monocular depth estimation, including traditional methods and deep learning-based methods. Recent advances in deep learning have enabled the development of powerful models that can estimate depth from a single image with high accuracy. Some of the most popular deep learning-based CNN architectures for monocular depth estimation are ResNet, DenseNet, MobileNet, VGG and U-Net. ResNet is a deep residual network that has demonstrated outstanding performance in various computer vision tasks. It is widely used for monocular depth estimation as it can learn a hierarchical representation of the input and avoid the vanishing gradient problem. DenseNet is a densely connected convolutional neural network that connects all layers to each other in a feedforward manner. This enables it to learn features from all layers and avoid the vanishing gradient problem. MobileNet is a lightweight CNN architecture that is designed specifically for mobile devices. It is a shallow network with depth-wise separable convolutional layers that reduces the computation and memory requirements. VGG is a deep CNN architecture that is known for its simplicity and high performance. It is widely used for various computer vision tasks including monocular depth estimation.

U-Net is a popular architecture for monocular depth estimation that has been widely used in the computer vision community. In obstacle avoidance, monocular depth estimation using U-Net can be used to provide depth information to a robot or autonomous vehicle, allowing it to detect obstacles and avoid collisions. By using a single camera, the system can estimate the depth of the scene in real-time, making it a cost-effective solution for obstacle avoidance.

One of the challenges of monocular depth estimation is the lack of ground truth data for training and evaluation. To train a U-Net model for monocular depth estimation, a large dataset of images with corresponding depth maps is required. One common dataset used for this task is the NYU Depth V2 dataset, which contains RGB-D images of indoor scenes. The accuracy of monocular depth estimation using U-Net can be affected by various factors, such as lighting conditions, object sizes, and distances. Therefore, it is important to train the model on a diverse set of images and to evaluate its performance in different scenarios to ensure its reliability in real-world applications.

6.1 NYU DEPTH V2 DATASET

The NYU-Depth V2 dataset is comprised of video sequences from a variety of indoor scenes as recorded by both the RGB and Depth cameras from the Microsoft Kinect. It features:

- 1449 densely labeled pairs of aligned RGB and depth images
- 464 new scenes taken from 3 cities
- 407,024 new unlabeled frames
- Each object is labeled with a class and an instance number

The dataset has several components:

- Labeled: A subset of the video data accompanied by dense multi-class labels. This data has also been preprocessed to fill in missing depth labels.
- Raw: The raw RGB, depth and accelerometer data as provided by the Kinect.
- Toolbox: Useful functions for manipulating the data and labels.

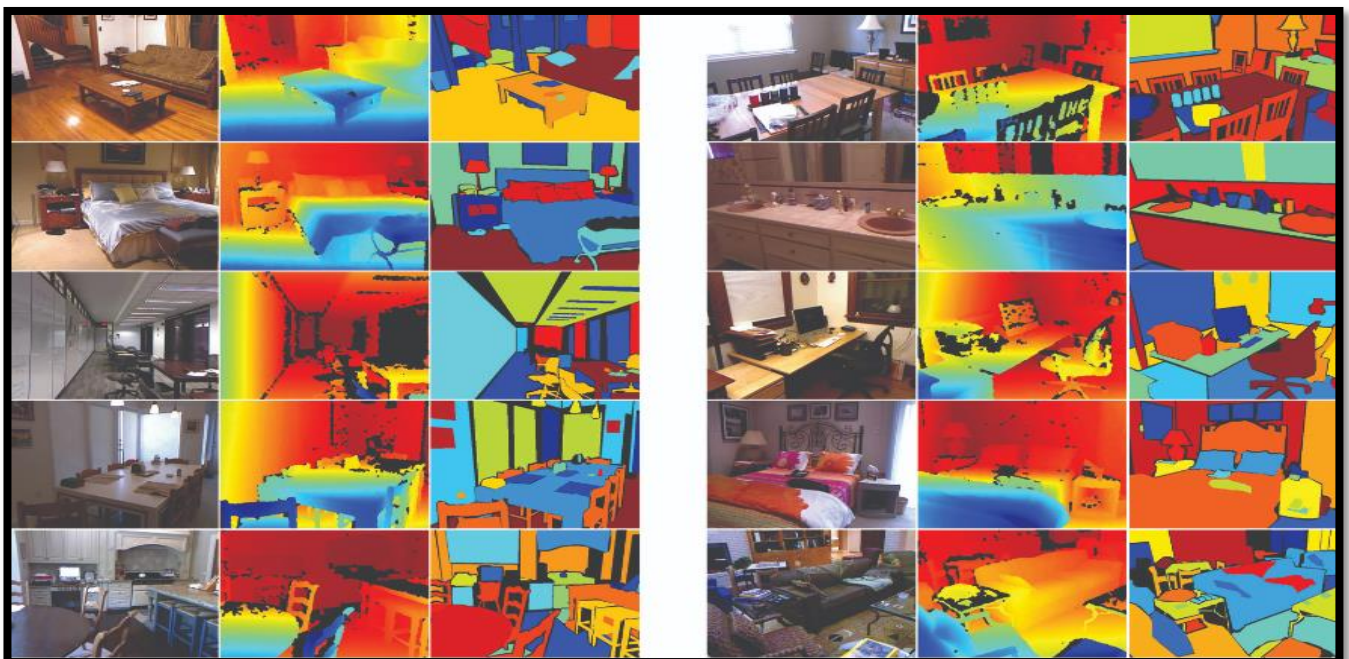


Fig 6.1 NYU Depth V2 Labeled Dataset

The above image shows the output from the RGB camera (left), pre-processed depth (center) and a set of labels (right) for the image. The labeled dataset is a subset of the Raw Dataset. It comprises of pairs of RGB and Depth frames that have been synchronized and annotated with dense labels for every image. In addition to the projected depth maps, we have included a set of preprocessed depth maps whose missing values have been filled in using the colorization scheme of Levin et al.

6.2 U-NET ARCHITECTURE

The U-Net architecture is a convolutional neural network (CNN) designed for image segmentation tasks. It consists of an encoding path and a decoding path, where the encoding path extracts high-level features from the input image, and the decoding path generates a segmentation mask that maps each pixel in the image to a class label. The encoding path of the U-Net architecture is composed of a series of convolutional and max-pooling layers that gradually reduce the spatial resolution of the input image while increasing the number of feature maps. Each convolutional layer applies a set of learnable filters to the input image, producing a set of output feature maps that capture different aspects of the input. The max-pooling layers downsample the feature maps, reducing the computational cost of the network and allowing the network to learn more robust features.

The decoding path of the U-Net architecture is composed of a series of up-convolutional and concatenation layers that gradually increase the spatial resolution of the feature maps while decreasing the number of feature maps. The up-convolutional layers perform an operation called transposed convolution or deconvolution, which upsamples the feature maps by inserting zeros between adjacent pixels and applying a set of learnable filters to interpolate the missing values.

The U-Net architecture also includes skip connections between the encoding and decoding paths, which enable the network to capture fine-grained details from the input image.

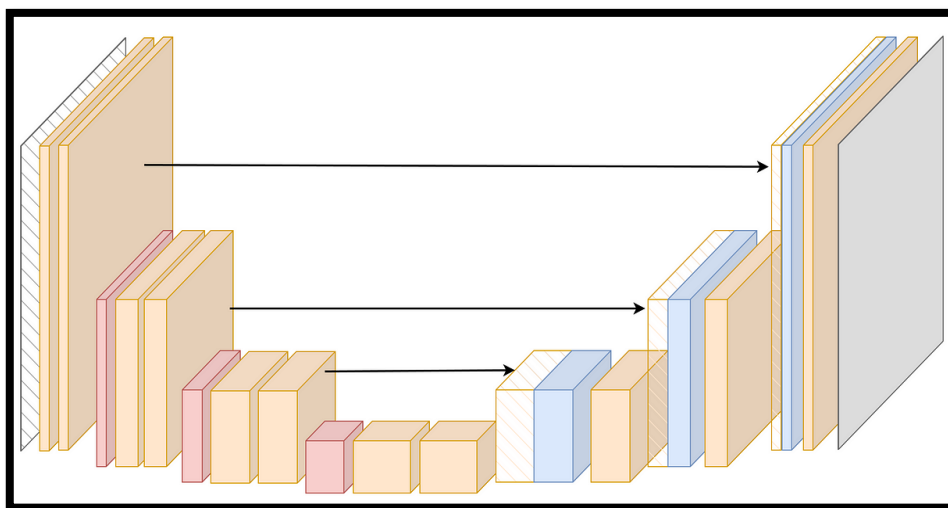


Fig 6.2 U-Net Architecture

7. IMPLEMENTATION

The objective of this project is to implement obstacle avoidance and goal detection using monocular depth estimation. The aim is to develop a system that can accurately estimate the depth of a scene from a single camera and use this information to navigate around obstacles and detect the location of a specified goal.

7.1 MONOCULAR DEPTH ESTIMATION

Monocular depth estimation is the task of estimating the depth value (distance relative to the camera) of each pixel given a single (monocular) RGB image. This project implements the U-Net Convolutional Neural Network with a DenseNet169 encoder on the NYU Depth V2 dataset.

Developing a Deep Learning model for Monocular Depth Estimation involves the following steps:

1. Data pre-processing:

The first step is to pre-process the NYU V2 depth data set. This involves resizing and normalizing the images, converting them to grayscale, and splitting the data set into training, validation, and testing sets.

2. Building the U-Net Model:

The U-Net model is a deep convolutional neural network architecture that has shown to be effective for image segmentation and depth estimation tasks. Our model utilized the Densenet169 architecture as the encoder, with trainable weights set to true for all layers. The decoder of our model consisted of 4 upsampling blocks, where each block was followed by a leaky relu activation function. The upsampling blocks were concatenated with the corresponding feature maps generated by the encoder, which allowed the model to learn more detailed and accurate depth features. The concatenation operation between the encoder feature maps and the decoder upsampling blocks is a key aspect of the U-Net architecture, as it helps to preserve the spatial information of the input image and ensures that the model can learn accurate depth information at various scales. By adding upsampling blocks, the model was able to upsample the feature maps generated by the encoder and learn to generate accurate depth maps. The use of leaky relu activation functions helped to prevent the vanishing gradient problem and allowed the model to learn more efficiently. Finally, the output layer of our model was a convolution layer with sigmoid activation and a 3x3 filter, which ensured that the output values of the model were between 0 and 1, as required for depth estimation tasks.

3. Defining the custom loss function:

The loss function calculates the difference between the ground truth and predicted depth maps y_i, \hat{y}_i respectively and measures how well the model is performing in estimating depth.

The loss function consists of three components: depth loss, edge loss, and structural similarity loss.

- a. The depth loss is calculated as the mean absolute error between the ground truth and predicted depth maps.

$$l_{\text{depth}}(y_i, \hat{y}_i) = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

- b. The edge loss component calculates the difference between the image gradients(d) of the predicted and ground truth depth maps. This helps to penalize the model for generating blurry or poorly defined edges in the depth map.

$$l_{\text{edges}}(y_i, \hat{y}_i) = \frac{1}{n} \sum_{i=1}^n (|dy(y_i, \hat{y}_i)| + |dx(y_i, \hat{y}_i)|)$$

- c. The structural similarity loss measures the structural similarity between the predicted and ground truth depth maps. This component is clipped to a range of 0 to 1 and is multiplied by a weight of 0.1.

$$l_{\text{ssim}}(y_i, \hat{y}_i) = \left[\frac{1}{2} (1 - \text{SSIM}(y_i, \hat{y}_i)) \right]$$

- d. The overall loss function is a weighted sum of the three components, where the weights for each component are set to 0.1, 1.0, and 1.0, respectively. The final loss is the sum of the three components weighted by their respective weights.

$$L(y_i, \hat{y}_i) = \lambda_{\text{depth}} l_{\text{depth}}(y_i, \hat{y}_i) + l_{\text{edges}}(y_i, \hat{y}_i) + l_{\text{ssim}}(y_i, \hat{y}_i)$$

By using this loss function, the model is optimized to generate depth maps that are sharp, structurally similar to the ground truth depth maps, and have accurate depth values.

4. Training the model:

To train our monocular depth estimation model, we used an AdamW optimizer with a learning rate of 0.0001, a weight decay of 1e-6. During training, we also implemented a polynomial learning rate decay function, which reduces the learning rate gradually after each epoch to help the model converge to a stable solution. The model was trained on the NYU Depth V2 training dataset and had about 42,828,569 parameters out of which trainable parameters were 42,663,929 and non-trainable parameters were 164,640.

5. Evaluating the model:

To evaluate the performance of our monocular depth estimation model, we split the NYU Depth V2 training dataset into training and validation sets using an 80:20 ratio. We used 80% of the dataset for training the model and the remaining 20% for evaluating the model's performance.

6. Testing the Model:

After training and validating our monocular depth estimation model on the NYU Depth V2 dataset, we tested the model's performance on the test set of the same dataset. During testing, we fed the test images into the model and obtained predicted depth maps as outputs. We then compared these predicted depth maps with the ground truth depth maps to compute evaluation metrics such as mean absolute error (MAE), root mean squared error (RMSE), mean squared error (MSE), Structural similarity (SSIM).

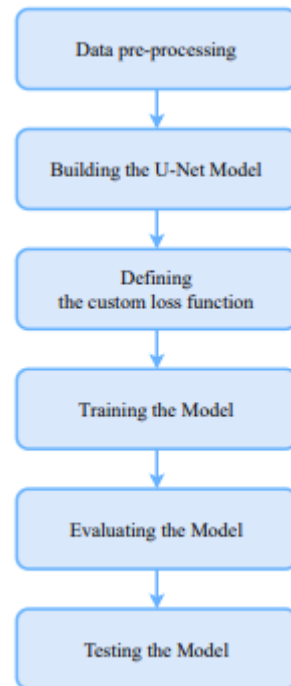


Fig 7.1 Flowchart for Monocular Depth Estimation

7.2 OBSTACLE AVOIDANCE

Using a depth image to segment the image into left, right, and center portions and using each portion to check for obstacles is a technique used for obstacle avoidance in this project. We have utilized a powerful combination of Robotic Operating System (ROS) middleware and Python programming language to create an effective obstacle avoidance framework.

Here are the general steps to use this approach:

1. Capture a depth image:

Use a depth sensor, such as a time-of-flight camera, a monocular camera or a stereo camera, to capture a depth image of the environment.

2. Preprocess the depth image:

Convert the raw depth data to a more usable format, such as a 2D matrix of depth values, and remove any noise or artifacts in the image.

3. Calculate normals to exclude the floor data:

To exclude the floor data in depth images, surface normals need to be calculated. This is achieved by calculating the gradients of the depth image in the x and y directions. These gradients are then used to compute the surface normals in the x, y, and z directions. The z component is set to a constant value of 1 for all pixels, as the camera is assumed to be pointing straight ahead. To identify the pixels that correspond to the floor, a threshold for the vertical angle is set. Pixels whose surface normals have an angle greater than this threshold are considered to correspond to the floor. This angle is computed by finding the angle between the surface normals and the vertical axis vector, which is a vector pointing straight up. Pixels whose surface normals have an angle greater than the threshold are then marked by a high value or a NaN value. This ensures that they are not detected as obstacles during the obstacle avoidance process.

4. Segment the image:

In order to enable effective obstacle avoidance, the depth image captured by the robot's camera can be divided into three segments: left, center, and right. By setting a threshold limit, the pixels that fall below this limit are marked as obstacles.

5. Check for obstacles:

For each remaining segment of the image, check for obstacles within a specified distance range. This can be done by thresholding the depth values or using clustering or segmentation algorithms to identify objects within the distance range.

6. Combine the obstacle information:

The obstacle information from each segment of the image is then combined to determine the overall obstacle-free area

7. Plan a safe path:

Based on this obstacle information, the robot can plan a safe path to follow, avoiding any obstacles that may be present.

8. Update the path:

Continuously update the path as the robot moves through the environment and new obstacles are detected. This can be done using real-time planning techniques.

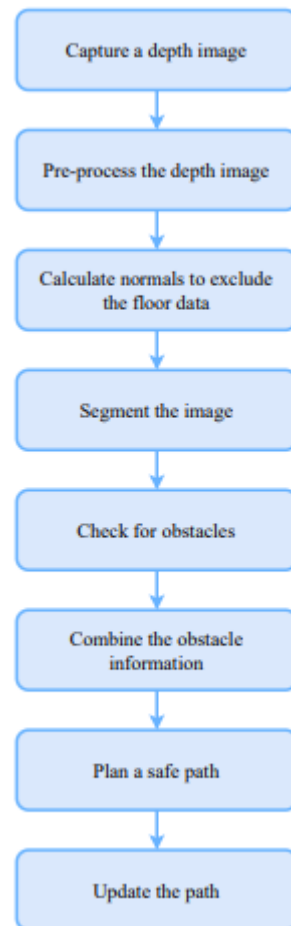


Fig 7.2 Flowchart for Obstacle Avoidance

7.3 GOAL DETECTION

In this project, Scale Invariant Feature Transform (SIFT) algorithm is utilized to detect the goal by extracting and matching distinctive keypoints from an image. SIFT works by analyzing the changes in an image at various scales and extracting features that are invariant to changes in scale, rotation, and lighting conditions. These features are known as keypoints, and they can be detected by identifying local maxima and minima in the Difference of Gaussian (DoG) scale space. Once keypoints are extracted from the robot's camera image, they are matched with the keypoints in the reference image of the goal using brute force matching.

Random Sample Consensus (RANSAC) algorithm is then applied to filter out the outliers and improve the accuracy of the keypoint matching. This allows the SIFT algorithm to determine the location and orientation of the goal relative to the robot's position.

After detecting the goal, a rectangle is drawn around it using perspective transform to determine its center position. This center position is then used to correct the robot's position by calculating the correct steering angles. This is done by taking the robot's camera image center and finding the difference between the center position of the goal and the center of the image. This difference is then used to calculate the steering angles required for the robot to reach the goal while avoiding obstacles.

There are other feature matching algorithms available for goal detection, such as Oriented fast and Rotated Brief (ORB) and Speeded Up Robust Features (SURF). ORB is faster and more efficient than SIFT, and is similar in terms of robustness to variations in lighting and orientation. However, it may not perform as well when it comes to detecting features accurately. Similarly, SURF is even faster than ORB, but it may struggle with changes in scale.

SIFT algorithm is chosen for this project as it offers a high level of robustness to variations in lighting conditions, scale, and orientation. This makes it ideal for detecting the goal from different viewpoints and under different lighting conditions. While ORB and SURF may be faster and more efficient, they may not provide the level of accuracy needed for our particular application.

8. HARDWARE AND SOFTWARE

8.1 TURTLEBOT

TurtleBot 2 is the world's most popular low cost, open source robot for education and research. This second generation personal robot is equipped with a powerful Kobuki robot base, a dual-core netbook, Orbbec Astra Pro Sensor and a Gyroscope.

- Size and Weight

DIMENSIONS	354 x 354 x 420 mm (14 x 14 x 16.5 in)
------------	--

WEIGHT	6.3 kg (13.9 lbs)
--------	-------------------

MAX PAYLOAD	5 kg (11 lbs)
-------------	---------------

- Speed and Performance

MAX SPEED	0.65 m/s (25.6 in/s)
-----------	----------------------

OBSTACLE CLEARANCE	15 mm (0.6 in)
--------------------	----------------

DRIVERS AND APIS	ROS
------------------	-----



Fig 8.1 TurtleBot 2

8.2 ROS

ROS stands for “Robot Operating System”. It is a powerful distributed computing framework tailored for robotics applications and it has been widely used. Each robotic task such as localization is hosted in a ROS node. ROS nodes can communicate with each other through topics and services. It is a great operating system for autonomous driving except that it suffers from several problems:

- (1) Reliability: ROS has a single master and no monitor to recover failed nodes.
- (2) Performance: When sending out broadcast messages, it duplicates the message multiple times, leading to performance degradation
- (3) Security: It has no authentication and encryption mechanisms.

We use ROS development tools and systems like Action Server, Launch Files, rViz Visualization, TF Transformations, Gazebo Simulation, etc.

8.3 GAZEBO

Gazebo is an open - source 3D robotics simulator. It has the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. It provides realistic rendering of environments including high-quality lighting, shadows, and textures. It can model sensors that "see" the simulated environment, such as laser range finders, cameras (including wide-angle), kinect style sensors, etc.

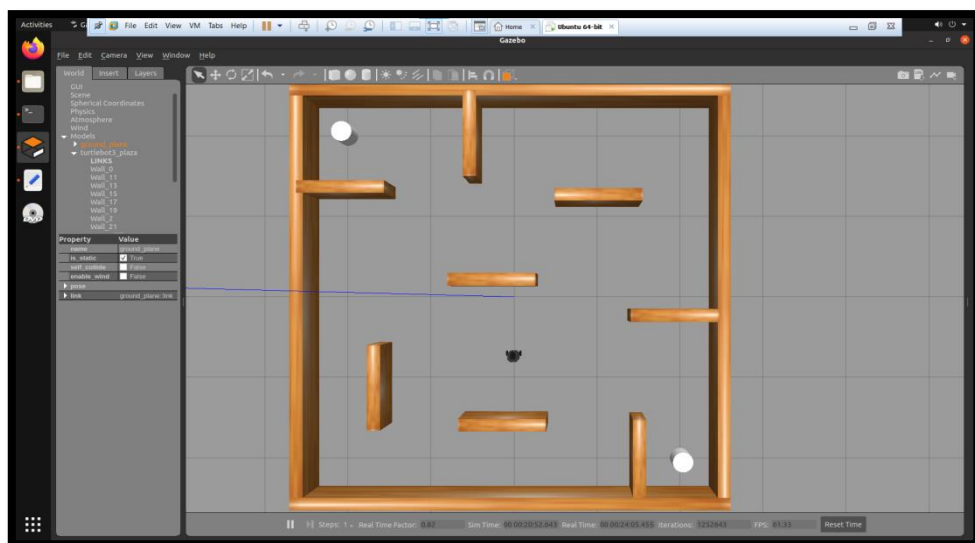


Fig 8.3 Gazebo

9. PROGRAMMING

9.1 OBSTACLE AVOIDANCE

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import Image
from std_msgs.msg import String
from cv_bridge import CvBridge, CvBridgeError
import os
import sys
import cv2
import numpy as np
import onnxruntime
from geometry_msgs.msg import Twist
from sensor_msgs.msg import Image as ImageMsg
import time
from matplotlib import pyplot

def interval_mapping(image, from_min, from_max, to_min, to_max):
    from_range = from_max - from_min
    to_range = to_max - to_min
    scaled = np.array((image - from_min) / float(from_range), dtype=float)
    return to_min + (scaled * to_range)

def image_callback(img):
    global is_computing, c, end
    c += 1
    if c % 38 == 0 or c == 1:
        rospy.loginfo("computing")
        rgb_image = np.frombuffer(img.data, dtype=np.uint8).reshape(img.height, img.width, -1)

        # Pre-processing the image
        img = cv2.resize(rgb_image, (640, 480)).astype(np.float32)
        scaled_img = (img - img.min()) / (img.max() - img.min())
        img = np.array(scaled_img)
```

```

input_name = session.get_inputs()[0].name
output_name = session.get_outputs()[0].name
img = img.reshape(1,480,640,3).astype(np.float32)
depth_image = session.run([output_name], {input_name: img})
depth_image = np.squeeze(depth_image, axis=0)
depth_image = np.squeeze(depth_image, axis=-1)
depth_image = np.squeeze(depth_image, axis=0)
depth_image = interval_mapping(depth_image, 0.0, 1.0, 0, 255).astype('uint8')

# Calculate gradient
gradient_x = cv2.Sobel(depth_image, cv2.CV_32F, 1, 0, ksize=3)
gradient_y = cv2.Sobel(depth_image, cv2.CV_32F, 0, 1, ksize=3)

# Compute surface normals
normals_x = -gradient_x / 255.0
normals_y = -gradient_y / 255.0
normals_z = np.ones_like(depth_image, dtype=np.float32)
norms = np.sqrt(normals_x**2 + normals_y**2 + normals_z**2)
normals_x /= norms
normals_y /= norms
normals_z /= norms

# Set threshold for vertical angle
theta_threshold = np.pi / 3 # 60 degrees

# Compute vertical axis vector
vertical_axis = np.array([0, 0, 1], dtype=np.float32)
vertical_axis = np.tile(vertical_axis, (depth_image.shape[0], depth_image.shape[1], 1))

# Compute angle between surface normals and vertical axis
angles = np.arccos(np.clip(np.abs(np.sum(normals_z[... , np.newaxis] * vertical_axis, axis=-1)), 0, 1))

# Identify pixels whose surface normals have an angle greater than the threshold
floor_mask = angles > np.deg2rad(theta_threshold)

# Set floor pixels to 255 depth value
depth_image[floor_mask] = 255
depth_image = interval_mapping(depth_image, 0, 255, 0.0, 1.0)

```

```

# Converting depth to meters
depth_image=depth_image*4.5

# Get the height and width of the image
height, width= depth_image.shape


# Split the image into left, center, and right parts
left = depth_image[:, :width//4]
center = depth_image[:, width//4:3*width//4]
right = depth_image[:, 3*width//4:]
left_half = left[height//2:, :]
mask1 = (left_half != 4.5)
center_third = center[height//2:, :]
mask2 = (center_third != 4.5)
right_half = right[height//2:, :]
mask3 = (right_half != 4.5)


# Threshold the depth image to detect obstacles in each region
threshold = 0.5
left_mask = np.where(left_half < threshold, 1, 0)
center_mask = np.where(center_third < threshold, 1, 0)
right_mask = np.where(right_half < threshold, 1, 0)


# Calculate the mean of the obstacle masks for each region
left_mean = np.mean(left_mask[mask1])
center_mean = np.mean(center_mask[mask2])
right_mean = np.mean(right_mask[mask3])


# If there is an obstacle in the center region, turn left or right depending on the available space
if center_mean > 0.2:
if left_mean < right_mean:
    rospy.loginfo('Turning left')
    twist.linear.x = 0.04
    twist.angular.z = 1
else:
    rospy.loginfo('Turning right')

```



```

        twist.linear.x = 0.04
        twist.angular.z = -1
    elif left_mean > 0.25:
        rospy.loginfo('Turning right')
        twist.linear.x = 0.0
        twist.angular.z = -0.5
    elif right_mean > 0.25:
        rospy.loginfo('Turning left')
        twist.linear.x = 0.0
        twist.angular.z = 0.5
    else:
        rospy.loginfo('straight')
        twist.linear.x = 0.04
        twist.angular.z = 0.0

    # Publish the velocity command
    is_computing = True
    cmd_vel_pub.publish(twist)
    rospy.sleep(0.2)
    is_computing = False
else:
    pass

def publish_small_linear_vel(cmd_vel_pub):
    twist = Twist()
    twist.linear.x = 0.04
    twist.angular.z = 0.0
    cmd_vel_pub.publish(twist)

if __name__ == '__main__':
    try:
        global c
        global is_computing, end
        end=0
        c=0
        is_computing=False

```

```

rospy.init_node('obstacle_avoidance')
cmd_vel_pub = rospy.Publisher('/cmd_vel_mux/input/navi', Twist, queue_size=10)
decision_pub = rospy.Publisher('/decision_topic', String, queue_size=10)
decision = "Start"
decision_pub.publish(decision)
image_sub = rospy.Subscriber("/usb_camera/image_raw", ImageMsg, image_callback, queue_size =
10)
twist = Twist()
rate = rospy.Rate(30)

# Load the ONNX model
path='/home/linux/Downloads/model_final.onnx'
session = onnxruntime.InferenceSession(path, providers=['CPUExecutionProvider'])
input_name = session.get_inputs()[0].name
output_name = session.get_outputs()[0].name
print("Starting Obstacle Avoidance")
while not rospy.is_shutdown():
    if not is_computing:
        publish_small_linear_vel(cmd_vel_pub)
    else:
        decision = "done"
        decision_pub.publish(decision)
    rate.sleep()
except KeyboardInterrupt:
    rospy.loginfo("Node terminated")
    del session
finally:
    rospy.loginfo("Node terminated")
    del session

```

9.2 GOAL DETECTION WITH OBSTACLE AVOIDANCE

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
import os
import sys
import cv2
import numpy as np
import onnxruntime
from sensor_msgs.msg import Image as ImageMsg
import time
from matplotlib import pyplot
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Point, Twist
from math import pow, atan2, sqrt, pi, ceil
import matplotlib.pyplot as plt

def interval_mapping(image, from_min, from_max, to_min, to_max):
    from_range = from_max - from_min
    to_range = to_max - to_min
    scaled = np.array((image - from_min) / float(from_range), dtype=float)
    return to_min + (scaled * to_range)

def odom_callback(msg):
    # Get current yaw angle of the robot
    global current_yaw
    orientation = msg.pose.pose.orientation
    q0, q1, q2, q3 = orientation.w, orientation.x, orientation.y, orientation.z
    siny_cosp = 2.0 * (q0 * q3 + q1 * q2)
    cosy_cosp = 1.0 - 2.0 * (q2 * q2 + q3 * q3)
    yaw = atan2(siny_cosp, cosy_cosp)
    current_yaw = yaw
def depth_callback(img):
    global is_computing, rotate, c, prev, check, cnt1
```

```

if Match==True or check==True or rotate==True or prev==True:
    return
cnt1+=1
if cnt1%10==0 or cnt1==1:
    rgb_image = np.frombuffer(img.data, dtype=np.uint8).reshape(img.height, img.width, -1)

# Pre-processing the image
img = cv2.resize(rgb_image,(640, 480)).astype(np.float32)
scaled_img = (img - img.min()) / (img.max() - img.min())
img=np.array(scaled_img)
input_name = session.get_inputs()[0].name
output_name = session.get_outputs()[0].name
img = img.reshape(1,480,640,3).astype(np.float32)
depth_image = session.run([output_name], {input_name: img})
depth_image = np.squeeze(depth_image, axis=0)
depth_image = np.squeeze(depth_image, axis=-1)
depth_image = np.squeeze(depth_image, axis=0)
depth_image = interval_mapping(depth_image, 0.0, 1.0, 0, 255).astype('uint8')

# Calculate gradient
gradient_x = cv2.Sobel(depth_image, cv2.CV_32F, 1, 0, ksize=3)
gradient_y = cv2.Sobel(depth_image, cv2.CV_32F, 0, 1, ksize=3)

# Compute surface normals
normals_x = -gradient_x / 255.0
normals_y = -gradient_y / 255.0
normals_z = np.ones_like(depth_image, dtype=np.float32)
norms = np.sqrt(normals_x**2 + normals_y**2 + normals_z**2)
normals_x /= norms
normals_y /= norms
normals_z /= norms

# Set threshold for vertical angle
theta_threshold = np.pi / 3.5

# Compute vertical axis vector
vertical_axis = np.array([0, 0, 1], dtype=np.float32)

```

```

vertical_axis = np.tile(vertical_axis, (depth_image.shape[0], depth_image.shape[1], 1))
# Compute angle between surface normals and vertical axis
angles = np.arccos(np.clip(np.abs(np.sum(normals_z[..., np.newaxis] * vertical_axis, axis=-1)), 0, 1))

# Identify pixels whose surface normals have an angle greater than the threshold
floor_mask = angles > np.deg2rad(theta_threshold)

# Set floor pixels to 255 depth value
depth_image[floor_mask] = 255
depth_image = interval_mapping(depth_image, 0, 255, 0.0, 1.0)

# Converting depth to meters
depth_image=depth_image*4.5

# Get the height and width of the image
height, width= depth_image.shape

# Split the image into left, center, and right parts
left = depth_image[:, :width//4]
center = depth_image[:, width//4:3*width//4]
right = depth_image[:, 3*width//4:]
left_half = left[height//2:, :]
mask1 = (left_half != 4.5)
center_third = center[height//2:, :]
mask2 = (center_third != 4.5)
right_half = right[height//2:, :]
mask3 = (right_half != 4.5)

# Threshold the depth image to detect obstacles in each region
threshold = 0.5
left_mask = np.where(left_half < threshold, 1, 0)
center_mask = np.where(center_third < threshold, 1, 0)
right_mask = np.where(right_half < threshold, 1, 0)

# Calculate the mean of the obstacle masks for each region
left_mean = np.mean(left_mask[mask1])
center_mean = np.mean(center_mask[mask2])
right_mean = np.mean(right_mask[mask3])

```

```

    if left_mean > 0.25:
        rospy.loginfo('right')
        twist.linear.x = 0.0
        twist.angular.z = -0.8
    elif right_mean > 0.25:
        rospy.loginfo('left')
        twist.linear.x = 0.0
        twist.angular.z = 0.8
    elif center_mean > 0.2:
        if left_mean < right_mean:
            rospy.loginfo('Turning left')
            twist.linear.x = 0.05
            twist.angular.z = 1
        else:
            rospy.loginfo('Turning right')
            twist.linear.x = 0.05
            twist.angular.z = -1
        else:
            rospy.loginfo('straight')
            twist.linear.x = 0.05
            twist.angular.z = 0.0
    c+=1
    is_computing = True
    cmd_vel_pub.publish(twist)
    rospy.sleep(0.2)
    is_computing = False
    if c==10:
        rotate=True
        c=0

```

```

else:

```

```

    pass

```

```

def image_callback(img):

```

```

    global rgb_image, counter, rotate, prev, check, percentage_match, count, cnt2, end, cc

```

```

    cnt2+=1

```

```

    if cnt2%10==0 or cnt2==1:

```

```

start=time.time()
rgb_image = np.frombuffer(img.data, dtype=np.uint8).reshape(img.height, img.width, -1)
global target_pixel, Match, goal_img,c,obstacle_detected,dst_pts_good,is_computing
global depth_image,is_computing, Match, c,turn_countl,turn_countr,prev
# Convert both images to grayscale
gray2 = cv2.cvtColor(rgb_image, cv2.COLOR_BGR2GRAY)
kp2, des2 = sift.detectAndCompute(gray2, None)
# Create matcher object
bf = cv2.BFMatcher()
# Match descriptors of the two images
match = bf.knnMatch(des1, des2, k=2)
matches = []
for m, n in match:
if m.distance < 0.6 * n.distance:
    matches.append(m)
if len(matches) < 20:
Match= False
print("No matches found in the image")
else:
# Apply RANSAC algorithm to filter out outliers
src_pts = np.float32([kp1[m.queryIdx].pt for m in matches]).reshape(-1, 1, 2)
dst_pts = np.float32([kp2[m.trainIdx].pt for m in matches]).reshape(-1, 1, 2)
# Calculate the homography matrix using RANSAC algorithm
M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
ht,wt = 240,432
points=np.float32([[0, 0], [0, ht], [wt, ht], [wt, 0]]).reshape(-1, 1, 2)
dst_points = cv2.perspectiveTransform(points, M)
dst_pts_good = dst_pts[mask.ravel() == 1]
homography = cv2.polylines(rgb_image, [np.int32(dst_points)], True, (255, 0, 0), 3)
dst_points=dst_points.reshape(4,2)
a = dst_points[0,0]
b = dst_points[1,1]
c = dst_points[0,1]
d = dst_points[3,0]
x,y = dst_points[0,0],dst_points[0,1]
w = int(d-a)

```

```

h = int(b-c)
matches = [matches[i] for i in range(len(matches)) if mask[i]]
print("2",len(matches))
if len(matches) > 15:
    Match= True
    rotate=False
    prev=True
    target_pixel = (int(x + w/2), int(y + h/2))
    num_good_matches = len(dst_pts_good)
    print(num_good_matches)
    print(len(kp1))
    total_matches = len(kp1)
    percentage_match = num_good_matches / total_matches * 100
    print("Percentage of match: {:.2f}%".format(percentage_match))
    print("Average matched point: ({}, {})".format(int(x + w/2), int(y + h/2)))
    rospy.loginfo("Moving towards Goal")
    plt.imshow(homography)
    img_n="image_{}.png".format(cc)
    img_path="/home/linux/Downloads/rec"
    cv2.imwrite(os.path.join(img_path,img_n),
    homography, [cv2.IMWRITE_PNG_COMPRESSION,0])
    cc+=1
    if (21 < percentage_match < 100):
        count += 1
        if count == 3:
            print("REACHED")
            end=True
        go_to_goal(target_pixel)
else:
    Match= False
    print("No matches found in the image")
    if Match==False and rotate==True:
is_computing=True
twist = Twist()
twist.angular.z = -0.7
cmd_vel_pub.publish(twist)

```



```

        counter+=1
    if counter==25:
        is_computing=False
        rotate=False
        end=time.time()
        print("time=", end-start)
    else:
        pass

def go_to_goal(target_pixel):
    global linear_velocity, angular_velocity, prev_yaw, turn_countr, turn_countl, is_computing, check
    x_diff = (target_pixel[0]) - image_width/2
    is_computing=True
    twist = Twist()
    if x_diff > 5:
        twist.angular.z = -angular_velocity
    elif x_diff < -5:
        twist.angular.z = angular_velocity
    else:
        twist.angular.z = 0.0
    twist.linear.x = linear_velocity
    cmd_vel_pub.publish(twist)
    rospy.sleep(0.2)
    is_computing=False
    check=check_obstacle()
    if check==True:
        rospy.loginfo('Moving Around Obstacle')
        prev_yaw=current_yaw
        move_around()
        while 1:
            rospy.loginfo('Reverting to previous angle')
            if turn_countr>turn_countl:
                twist.linear.x =0.0
                twist.angular.z = 0.2
            else:
                twist.linear.x = 0.0

```

```

    twist.angular.z = -0.2
time.sleep(0.5)
is_computing = True
cmd_vel_pub.publish(twist)
is_computing = False
if abs(prev_yaw-current_yaw) < 0.05:
    twist.linear.x = 0.0
    twist.angular.z = 0.0
    cmd_vel_pub.publish(twist)
    break

```

```
def move_around():
```

```
    while 1:
```

```

        global rgb_image,prev_yaw,turn_countr,turn_countl,is_computing
        is_computing = False
        img = cv2.resize(rgb_image,(640, 480)).astype(np.float32)
        scaled_img = (img - img.min()) / (img.max() - img.min())
        img=np.array(scaled_img)
        input_name = session.get_inputs()[0].name
        output_name = session.get_outputs()[0].name
        img = img.reshape(1,480,640,3).astype(np.float32)
        depth_image = session.run([output_name], {input_name: img})
        depth_image = np.squeeze(depth_image, axis=0)
        depth_image = np.squeeze(depth_image, axis=-1)
        depth_image = np.squeeze(depth_image, axis=0)
        depth_image = interval_mapping(depth_image, 0.0, 1.0, 0, 255).astype('uint8')

```

Calculate gradient

```

gradient_x = cv2.Sobel(depth_image, cv2.CV_32F, 1, 0, ksize=3)
gradient_y = cv2.Sobel(depth_image, cv2.CV_32F, 0, 1, ksize=3)

```

Compute surface normals

```

normals_x = -gradient_x / 255.0
normals_y = -gradient_y / 255.0
normals_z = np.ones_like(depth_image, dtype=np.float32)
norms = np.sqrt(normals_x**2 + normals_y**2 + normals_z**2)
normals_x /= norms

```

```

normals_y /= norms
normals_z /= norms

# Set threshold for vertical angle
theta_threshold = np.pi / 3.5

# Compute vertical axis vector
vertical_axis = np.array([0, 0, 1], dtype=np.float32)
vertical_axis = np.tile(vertical_axis, (depth_image.shape[0], depth_image.shape[1], 1))

# Compute angle between surface normals and vertical axis
angles = np.arccos(np.clip(np.abs(np.sum(normals_z[..., np.newaxis] * vertical_axis, axis=-1)), 0,
1))

# Identify pixels whose surface normals have an angle greater than the threshold
floor_mask = angles > np.deg2rad(theta_threshold)

# Set floor pixels to 255 depth value
depth_image[floor_mask] = 255
depth_image = interval_mapping(depth_image, 0, 255, 0.0, 1.0)

# Converting depth to meters
depth_image=depth_image*4.5

# Get the height and width of the image
height, width= depth_image.shape

# Split the image into left, center, and right parts
left = depth_image[:, :width//4]
center = depth_image[:, width//4:3*width//4]
right = depth_image[:, 3*width//4:]
left_half = left[height//2:, :]
mask1 = (left_half != 4.5)
center_third = center[height//2:, :]
mask2 = (center_third != 4.5)
right_half = right[height//2:, :]
mask3 = (right_half != 4.5)

# Threshold the depth image to detect obstacles in each region
threshold = 0.5
left_mask = np.where(left_half < threshold, 1, 0)

```

```
center_mask = np.where(center_third < threshold, 1, 0)
right_mask = np.where(right_half < threshold, 1, 0)
```

Calculate the mean of the obstacle masks for each region

```
left_mean = np.mean(left_mask[mask1])
center_mean = np.mean(center_mask[mask2])
right_mean = np.mean(right_mask[mask3])
```

```
if left_mean > 0.25:
```

```
    rospy.loginfo('right')
    twist.linear.x = 0.0
    twist.angular.z = -0.8
    turn_countr+=1
```

```
elif right_mean > 0.25:
```

```
    rospy.loginfo('left')
    twist.linear.x = 0.0
    twist.angular.z = 0.8
    turn_countl+=1
```

```
elif center_mean > 0.2:
```

```
    if left_mean < right_mean:
```

```
        rospy.loginfo('Turning left')
        twist.linear.x = 0.05
        twist.angular.z = 1
        turn_countl+=1
```

```
    else:
```

```
        rospy.loginfo('Turning right')
        twist.linear.x = 0.05
        twist.angular.z = -1
        turn_countr+=1
```

```
    else:
```

```
        break
```

```
is_computing = True
```

```
cmd_vel_pub.publish(twist)
```

```
rospy.sleep(0.2)
```

```
is_computing = False
```

```
def check_obstacle():
```

```
    global rgb_image, linear_velocity, angular_velocity, is_computing
```

```

is_computing = False
img = cv2.resize(rgb_image,(640, 480)).astype(np.float32)
scaled_img = (img - img.min()) / (img.max() - img.min())
img=np.array(scaled_img)
input_name = session.get_inputs()[0].name
output_name = session.get_outputs()[0].name
img = img.reshape(1,480,640,3).astype(np.float32)
depth_image = session.run([output_name], {input_name: img})
depth_image = np.squeeze(depth_image, axis=0)
depth_image = np.squeeze(depth_image, axis=-1)
depth_image = np.squeeze(depth_image, axis=0)
depth_image = interval_mapping(depth_image, 0.0, 1.0, 0, 255).astype('uint8')

```

Calculate gradient

```

gradient_x = cv2.Sobel(depth_image, cv2.CV_32F, 1, 0, ksize=3)
gradient_y = cv2.Sobel(depth_image, cv2.CV_32F, 0, 1, ksize=3)

```

Compute surface normals

```

normals_x = -gradient_x / 255.0
normals_y = -gradient_y / 255.0
normals_z = np.ones_like(depth_image, dtype=np.float32)
norms = np.sqrt(normals_x**2 + normals_y**2 + normals_z**2)
normals_x /= norms
normals_y /= norms
normals_z /= norms

```

Set threshold for vertical angle

```

theta_threshold = np.pi / 3.5

```

Compute vertical axis vector

```

vertical_axis = np.array([0, 0, 1], dtype=np.float32)
vertical_axis = np.tile(vertical_axis, (depth_image.shape[0], depth_image.shape[1], 1))

```

Compute angle between surface normals and vertical axis

```

angles = np.arccos(np.clip(np.abs(np.sum(normals_z[..., np.newaxis] * vertical_axis, axis=-1)), 0, 1))

```

Identify pixels whose surface normals have an angle greater than the threshold

```

floor_mask = angles > np.deg2rad(theta_threshold)

```

Set floor pixels to 255 depth value

```
depth_image[floor_mask] = 255
```

```
depth_image = interval_mapping(depth_image, 0, 255, 0.0, 1.0)
```

```
depth_image=depth_image*4.5
```

Get the height and width of the image

```
height, width= depth_image.shape
```

Split the image into left, center, and right parts

```
left = depth_image[:, :width//4]
```

```
center = depth_image[:, width//4:3*width//4]
```

```
right = depth_image[:, 3*width//4:]
```

```
left_half = left[height//2:, :]
```

```
mask1 = (left_half != 4.5)
```

```
center_third = center[height//2:, :]
```

```
mask2 = (center_third != 4.5)
```

```
right_half = right[height//2:, :]
```

```
mask3 = (right_half != 4.5)
```

Threshold the depth image to detect obstacles in each region

```
threshold = 0.5
```

```
left_mask = np.where(left_half < threshold, 1, 0)
```

```
center_mask = np.where(center_third < threshold, 1, 0)
```

```
right_mask = np.where(right_half < threshold, 1, 0)
```

Calculate the mean of the obstacle masks for each region

```
left_m = np.mean(left_mask[mask1])
```

```
center_m = np.mean(center_mask[mask2])
```

```
right_m= np.mean(right_mask[mask3])
```

If there is an obstacle in the center region, turn left or right depending on the available space

```
if center_m > 0.1:
```

```
    return True
```

```
elif left_m > 0.1:
```

```
    return True
```

```
elif right_m > 0.1:
```

```
    return True
```

```

else:
    return False
def publish_small_linear_vel(cmd_vel_pub):
    twist = Twist()
    twist.linear.x = 0.04
    cmd_vel_pub.publish(twist)

if __name__ == '__main__':
    try:
        global counter,turn_countl,turn_countr, linear_velocity,
angular_velocity,dst_pts_good,rotate,cnt1,cnt2,end
        global
is_computing,Match,goal_img,obstacle_detected,x,percentage_match,initial_yaw,c,prev,check,cc
        cc=0
        end=False
        check=False
        prev=False
        percentage_match=0
        cnt1=0
        cnt2=0
        c=0
        count=0
        goal_img=cv2.imread("/home/linux/Downloads/image_goal_final.png")

# Create SIFT detector
        sift = cv2.xfeatures2d.SIFT_create()
        gray1 = cv2.cvtColor(goal_img, cv2.COLOR_BGR2GRAY)

# Detect keypoints and compute descriptors for both images
        kp1, des1 = sift.detectAndCompute(gray1, None)
        dst_pts_good=[]
        Match=False
        rotate=True
        obstacle_detected=False
        initial_yaw=0.0
        counter=0
        turn_countl=0

```

```

turn_countr=0
is_computing=False
current_yaw = 0.0
linear_velocity = 0.04
angular_velocity = 0.3
image_width = 432
image_height = 240
rospy.init_node('obstacle_avoidance')
image_sub = rospy.Subscriber("/usb_camera/image_raw", ImageMsg, image_callback)
image_sub1 = rospy.Subscriber("/usb_camera/image_raw", ImageMsg, depth_callback)
cmd_vel_pub = rospy.Publisher('/cmd_vel_mux/input/navi', Twist, queue_size=10)
odom_sub = rospy.Subscriber('/odom', Odometry, odom_callback)
initial_yaw=current_yaw
twist = Twist()
rate = rospy.Rate(10)

```

Load the ONNX model

```

path='/home/linux/Downloads/model_final.onnx'
session = onnxruntime.InferenceSession(path, providers=['CPUExecutionProvider'])
input_name = session.get_inputs()[0].name
output_name = session.get_outputs()[0].name
print("Starting Obstacle Avoidance")

```

Continuously publish the small linear velocity until the callback function starts computing

```

while not rospy.is_shutdown():

```

```

    if not is_computing:
        publish_small_linear_vel(cmd_vel_pub)
    elif end==True:
        twist = Twist()
        twist.linear.x = 0.0
        twist.angular.z=0.0
        cmd_vel_pub.publish(twist)
        break
    rate.sleep()

```

```

rospy.shutdown(image_callback)

```

```

except KeyboardInterrupt:

```

```

    del session

```



```
sys.exit("Node terminated")
```

```
finally:
```

```
del session
```

```
sys.exit("Node terminated")
```

10. RESULTS AND DISCUSSION

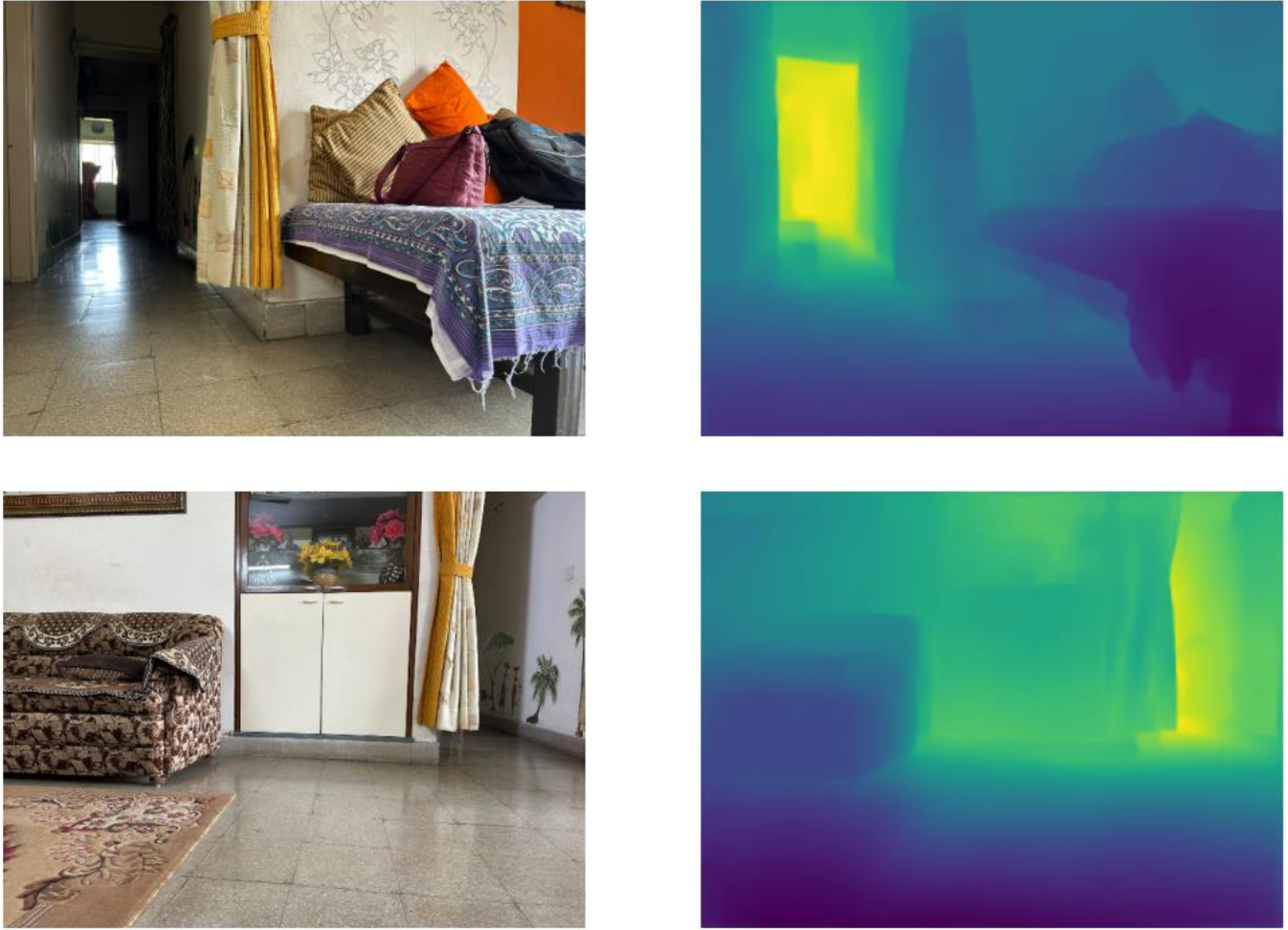


Fig 10.1 Input RGB images and its predicted Depth maps

The above output shows the predicted depth maps of the input RGB images. The color map shown in the output is “Viridis” from Matplotlib color maps. We can infer that brighter the color, the farther away is the object and darker the color, the nearer is the object.

The monocular depth estimation model was trained on NYU V2 depth dataset to accurately estimate the distance to objects in the robot's field of view. The ONNX model was optimized for deployment on the robot's hardware, allowing for real-time inference with low computational overhead.

The use of machine learning for obstacle avoidance in robotics has become increasingly popular in recent years due to its ability to handle complex and dynamic environments. The monocular depth estimation model used in this project is a powerful tool for accurately detecting obstacles in the robot's environment, making it an ideal choice for obstacle avoidance.

One of the key advantages of using an ONNX model is its compatibility with a wide range of hardware platforms, allowing for easy deployment on a variety of robots with different specifications. This makes it a highly scalable solution that can be easily adapted for use in different scenarios and applications.

However, one potential limitation of using machine learning for obstacle avoidance is the need for large amounts of training data to ensure accurate predictions. Additionally, the model may not be able to handle all types of obstacles, such as transparent or reflective objects, which may require additional sensors or techniques to detect.

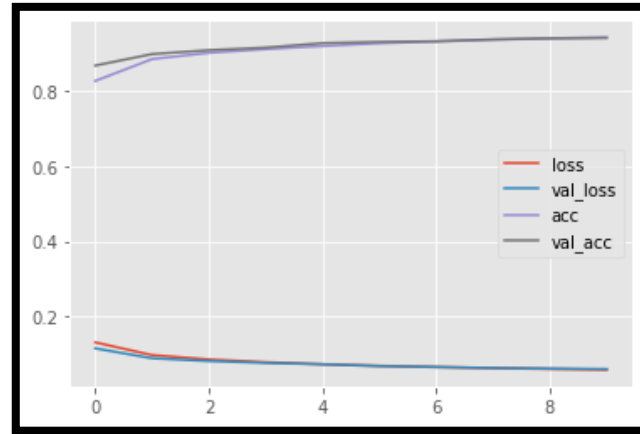


Fig 10.2 Accuracy and Loss graph of Training vs Validation

The above training versus validation accuracy and loss graph shows that the model achieved high accuracy and low loss on both training and validation datasets, indicating that the model did not overfit or underfit the data.

```
[14]: model.evaluate(val_generator)

1267/1267 [=====] - 389s 296ms/step - loss: 0.0430 - depth_acc: 0.9699 - structural_similarity_index: 0.9381 - rmse: 0.0406 - mae: 0.0252 - mse: 0.0018

[14]: [0.0430380180478096,
0.9699313044548035,
0.9380631446838379,
0.04057205468416214,
0.025227569043636322,
0.001771802082657814]

[15]: model.evaluate(test_generator)

40/40 [=====] - 40s 880ms/step - loss: 0.0946 - depth_acc: 0.8583 - structural_similarity_index: 0.8482 - rmse: 0.1327 - mae: 0.0997 - mse: 0.0181

[15]: [0.09457661211490631,
0.8582652807235718,
0.8482334017753601,
0.13273675739765167,
0.09974810481071472,
0.018101457506418228]
```

Fig 10.3 Performance metrics of the Model

The image above illustrates the performance metrics of the monocular depth estimation model on both the validation and test datasets. Specifically, the validation dataset, which was derived from an 80/20 split of the NYU Depth V2 training dataset, consisted of 10,138 images and was processed in batches of 8.

Conversely, the test dataset consisted of 654 images from the NYU Depth V2 test dataset and was processed in batches of 16.

The table below summarizes the performance of our model on the validation and test datasets. The evaluation metrics used are Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), Mean Squared Error (MSE), Structural Similarity Index (SSIM) and Depth accuracy. By measuring RMSE, MAE, MSE, SSIM and depth accuracy, we can determine how well the algorithm performed and identify areas for improvement.

- Root Mean Squared Error (RMSE)

RMSE is a commonly used metric to evaluate the accuracy of a regression model, particularly in depth estimation models. It measures the average distance between the predicted values and the actual values. RMSE is calculated by taking the square root of the average of the squared differences between the predicted and actual values. A lower RMSE indicates that the model has better accuracy in predicting the depth values.

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

- Mean Absolute Error (MAE)

MAE is another measure of the difference between the predicted and ground truth depth values. It calculates the average absolute difference between the two sets of values. A lower MAE also indicates better accuracy.

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

- Mean Squared Error (MSE)

MSE is similar to RMSE, but it calculates the average of the squared difference between predicted and actual depth values. Like RMSE, a lower MSE indicates better accuracy.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

where,

y_i : True values in depth image

\hat{y}_i : Redicted values in depth image

n: Number of samples in the dataset

- Structural similarity index (SSIM):

SSIM is a metric used to evaluate the similarity between two images. It takes into account several factors, including luminance, contrast, and structure, and can be used to measure the similarity between the predicted and ground truth depth maps.

- Depth accuracy:

Depth accuracy is a measure of the accuracy of the estimated depth values. It is typically calculated as the percentage of pixels in the depth map that are within a certain tolerance of the ground truth depth values. A higher depth accuracy indicates better performance.

Metrics	Validation Set	Testing Set
RMSE	0.04057205468416214	0.13273675739765167
MAE	0.025227569043636322	0.09974810481071472
MSE	0.001771802082657814	0.018101457506418228
SSIM	0.9380631446838379	0.8482334017753601
Depth Accuracy	0.9699313044548035	0.8582652807235718

Table 10.1 Performance analysis of the Model

The depth accuracy here is calculated by rounding the predicted and ground truth depth values to the nearest value and then comparing them. A value of 1 means the predicted depth value matches the ground truth depth value, and a value of 0 means they do not match. The average of these 1's and 0's across all pixels in the predicted depth map gives us the depth accuracy metric. The model achieved an RMSE of 0.0405 on the validation set and 0.1327 on the test set. The MAE values were 0.025 and 0.099 on the validation and test datasets, respectively. The MSE was 0.0017 and 0.0181 on the validation and test sets, respectively. In terms of structural similarity index, the model achieved a score of 0.938 and 0.8482 on the validation and test sets, respectively. The depth accuracy of the model was 97% and 86% on the validation and test datasets, respectively. These results demonstrate that our model achieved good performance on both validation and test datasets, with relatively low errors and good accuracy.



Fig 10.4 Robot in an Environment with obstacles

The images above depict a Kobuki robot navigating through an environment filled with obstacles while successfully avoiding them along the way. The robot's obstacle avoidance system was tested in a variety of indoor environments settings with varying obstacle sizes and shapes.



Fig 10.5 Estimated Goal Position

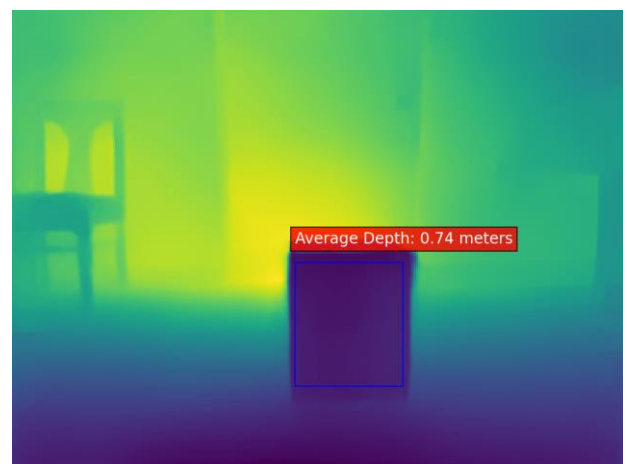
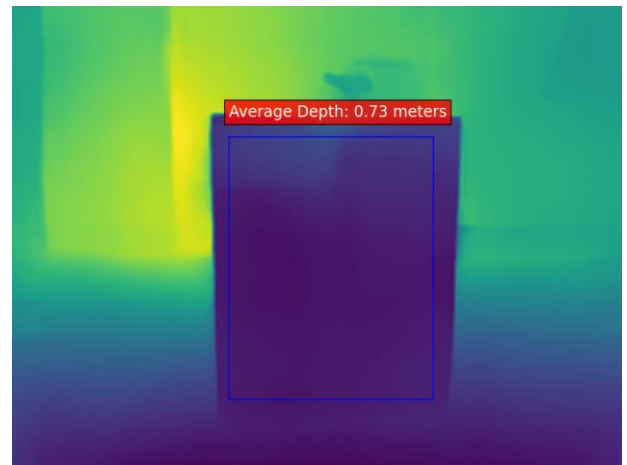
A rectangle representing the estimated position of the goal is visible in the above image. This is calculated using perspective transform applied to the robot's view image.

10.1 ERROR ANALYSIS

Our model predicted depth images in 8-bit format. To establish measurements in physical units, we converted the images to meters using the depth range capabilities of the Kinect camera, which was also used in the NYU Depth V2 dataset. The Kinect camera's depth range was 0.5 to 4.5 meters. After normalizing the predicted depth images to a range of 0 to 1, we applied a conversion process using a scaling factor of 4 and a minimum range of 0.5 meters, based on the depth range of the Kinect camera. These steps ensured standardized and accurate depth measurements for our analysis.

To estimate the accuracy of our model, we conducted an error analysis on objects at varying distances at the ground as well as elevated objects at different distances. Our approach to predicting object distance involved calculating the average of all pixels in the predicted depth image that corresponded to the object of interest. This method allowed us to obtain a reliable estimate of object distance and facilitated our error analysis for both ground-level and elevated objects at different distances.

▪ For object on ground



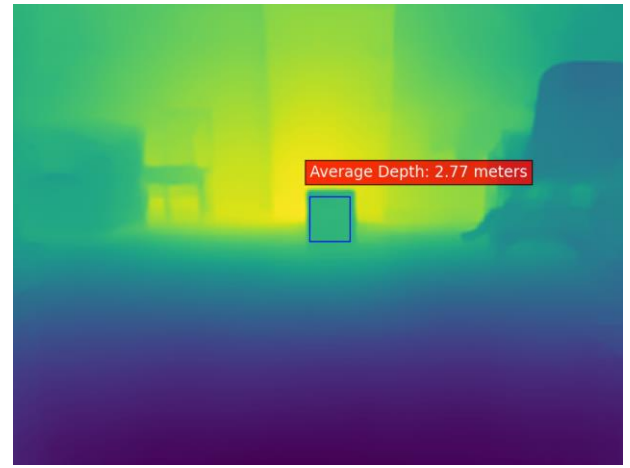
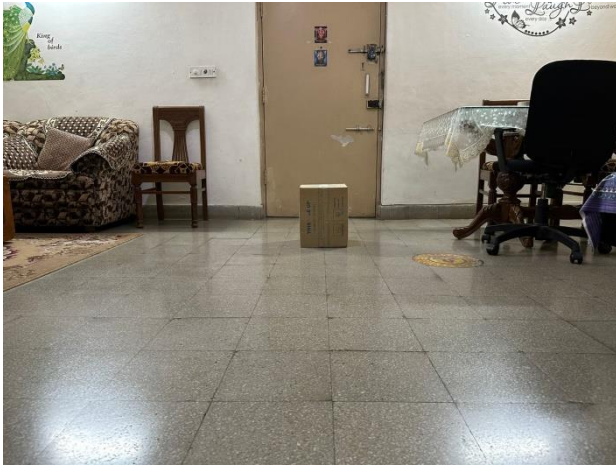
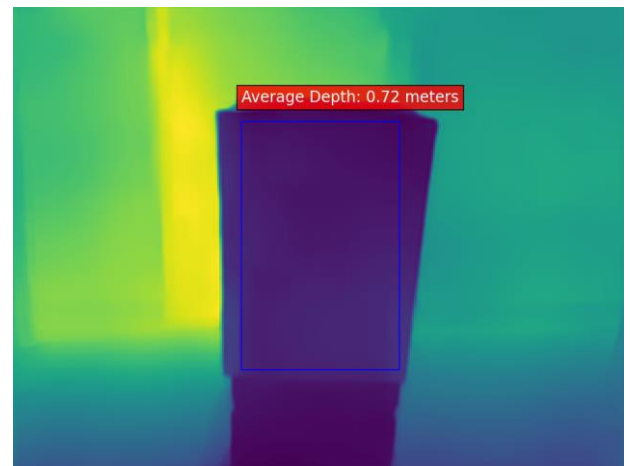


Fig 10.6 Depth predictions for Ground object

Ground Truth Value	Predicted Value	Error
0.5 m	0.73 m	46 %
1 m	0.74 m	26 %
1.5 m	1.65 m	10 %
2 m	2.31 m	15.5 %
2.5 m	2.77 m	10.8 %

Table 10.2 Error analysis for Ground object

- For object at a height of 50 cm



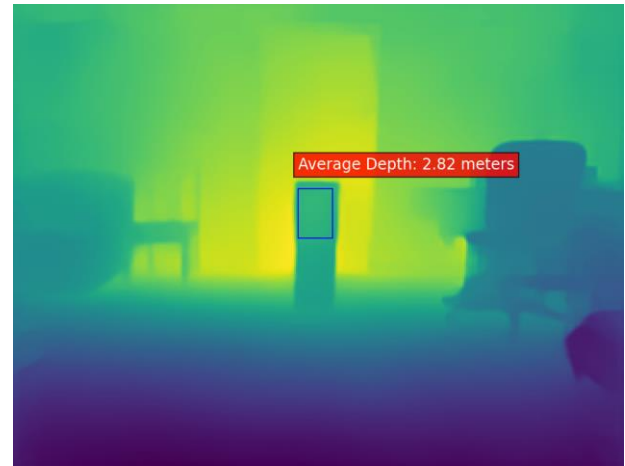
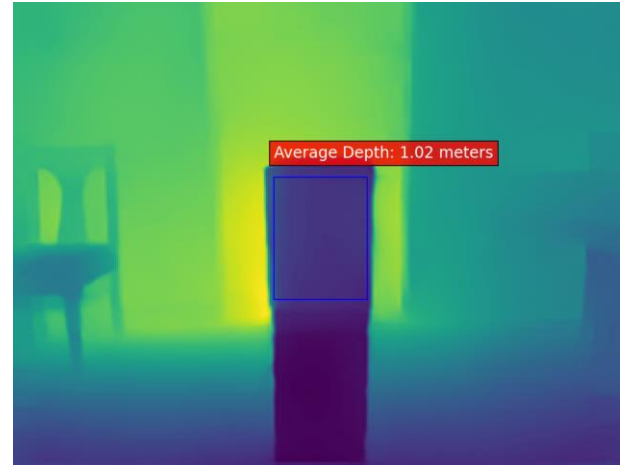


Fig 10.7 Depth predictions for Elevated object

Ground Truth Value	Predicted Value	Error
0.5 m	0.72 m	44 %
1 m	1.02 m	2 %
1.5 m	1.68 m	12 %
2 m	1.90 m	5 %
2.5 m	2.82 m	12.8 %

Table 10.3 Error analysis for Elevated object

Upon evaluating our model's performance, we discovered that it exhibited optimal results for object distances of 1 meter or greater. However, we also identified a decrease in performance for objects located at closer ranges of 0.5 meters or less. Our analysis further revealed that our model was better suited for predicting the distances of elevated objects in comparison to those at ground level. We attribute the

observed decrease in performance for closer ranges and ground-level objects to the lack of available close-range and low-lying data within the NYU Depth V2 dataset.

In conclusion, our evaluation indicates that the monocular depth estimation model we developed performs best for objects located at distances of 1 meter or greater, where high accuracy is required. However, for applications such as obstacle avoidance where the primary concern is detecting the presence of obstacles rather than accurately measuring their distance, the model may still be suitable even for objects located at closer ranges, despite the observed decrease in performance. Nonetheless, it is important to acknowledge that the model's accuracy may vary depending on the application and distance range, and further improvements and enhancements are needed to address its limitations and expand its potential range of applications.

11. TROUBLESHOOTING

11.1 PROBLEMS FACED

- Problem 1:

Limited hardware resources can slow down the model's inference speed, resulting in delayed obstacle detection and avoidance. CPUs have fewer cores, lower memory bandwidth and limited cache memory, which can lead to serial computation and slower data transfer and computation.

- Problem 2:

Object detection and avoidance models trained on labelled datasets may not generalize well to all real-world scenarios, especially those with large gaps or occluded obstacles.

- Problem 3:

Using an ONNX model for obstacle detection and avoidance on the Turtlebot 2 Kobuki robot can increase energy consumption.

- Problem 4:

Detecting objects that are in close proximity to the Turtlebot 2 Kobuki robot can pose a challenge due to a number of factors, including the range of the training data set, sensor range and resolution, robot speed and processing time. The NYU V2 depth dataset, which uses Microsoft Kinect sensors, has a range of 0.5 to 4.5 meters, making it difficult to detect objects that are too close. Moreover, if the robot is moving too fast, it may not have sufficient time to detect and avoid obstacles, particularly if the detection algorithm is computationally intensive.

- Problem 5:

Detecting a goal image while avoiding and detecting obstacles is a challenging task that requires performing complex processes simultaneously. This includes object detection, recognition and navigation, each of which requires significant computational resources and specialized algorithms. The robot needs to make complex decisions in real-time based on its sensory inputs, requiring advanced decision-making algorithms and a high-level understanding of its environment and goals.

- Problem 6:

Obstacle detection and avoidance systems can face challenges when dealing with reflective surfaces, such as mirrors or highly reflective floors. The reflections can create false positives and false negatives, leading to inaccurate detection and avoidance.

- Problem 7:

Several networking issues can cause delays or drops in communication, including network congestion, latency, signal interference, and bandwidth limitations.

- Problem 8:

When a model is trained on well-lit images and tested on dark or low-light images, it may not perform

well due to the difference in lighting conditions. This can cause the model to make mistakes or not recognize objects accurately.

- Problem 9:

When a robot moves on a rough surface, it can experience slippage or skidding, causing the actual movement to deviate from the intended movement specified in the program.

- Problem 10:

The height difference between the data in the NYU V2 depth dataset and the physical position of the robot can cause incorrect predictions. The dataset's height measurements may not accurately represent the robot's low-level perspective, leading to errors in object detection and avoidance.

- Problem 11:

A potential issue that can arise in depth images is the floor data being identified as an obstacle.

11.2 SOLUTION TO THE PROBLEMS

- Problem 1:

To address the limitations, optimizing the ONNX model and software for CPU, using hardware accelerators like GPUs or TPUs and slowing down the robot's speed are recommended.

- Problem 2:

To improve the model's performance, collecting additional training data that includes these challenging scenarios and fine-tuning the model using transfer learning or other techniques can be used.

- Problem 3:

To address the issue, we can reduce the runtime of the robot. Optimizing the model for low-power consumption by reducing its size and complexity, using more efficient algorithms or techniques, or specialized low-power hardware can help minimize the impact on energy consumption.

- Problem 4:

To tackle the problem, one solution could be to slow down the robot's speed, while another approach could be to train the model using images of objects that are closer in proximity to the robot.

- Problem 5:

To overcome the challenges, the algorithms must be carefully designed and optimized along with optimized hardware, and decision-making capabilities of the robot.

- Problem 6:

To overcome the issues, one approach is to train the model to detect and handle reflective surfaces. Alternatively, less reflective sensors such as LIDAR or ultrasound can be used, or algorithms specifically designed for reflective surfaces can be employed.

- Problem 7:

Offloading some processing tasks from the main computer to an edge device, such as Jetson and Raspberry Pi, can help reduce the latency and improve overall performance by enabling real-time processing of sensor data and other inputs.

- Problem 8:

To address the issue one way is to retrain the model using images captured under different lighting conditions or use image processing techniques to enhance images captured in dark or low-light conditions.

- Problem 9:

To overcome the problem the robot's actual movement needs to be monitored in real-time and compared with the intended movement specified in the program. This can be achieved by using various sensors such as wheel encoders, IMUs, and force sensors. One effective way to address this issue is to use feedback control techniques to adjust the robot's movement based on the sensor readings. Additionally in our case camera sensor data can be used to validate if the correct direction was taken and correct any deviations in real-time.

- Problem 10:

To mitigate the issue one approach is to adjust the camera's mounting height or angle to align with the robot's position. Additionally, collecting more training data from the robot's actual viewpoint can improve the model's accuracy.

- Problem 11:

To resolve the issue, one possible solution is to remove the floor data by fitting a normal to the plane of the floor and setting it to a NaN value. This approach can help ensure that the robot's obstacle detection and avoidance systems do not mistake the floor as an obstacle and instead focus on detecting actual obstacles.

12. APPLICATIONS

There are a variety of benefits mobile robots can provide in indoor environments, such as in warehouses or factories; they could be used for tasks such as material handling or surveillance. Obstacle avoidance is a critical function for mobile robots in these environments. Some potential applications of mobile robots in indoor obstacle avoidance are as follows:

- **Search and Rescue Operation:**

When conducting an indoor search and rescue operation, it is not always possible to have a full map of the environment. In such situations, mobile robots can still be used effectively by employing techniques such as image-based navigation and obstacle avoidance. Mobile robots can be programmed to navigate towards a given image, such as a picture of a missing person or a specific landmark in the search area while avoiding obstacles in its path.

- **Automated Inventory Management:**

Mobile robots can be used to autonomously navigate through warehouses and other facilities to scan and identify items. By using obstacle avoidance technology, Mobile robots can navigate around obstacles such as shelving units and other inventory items, helping to reduce the time and effort required for inventory management.

- **Surveillance and Security:**

Mobile robots equipped with cameras and other sensors can be used to monitor indoor environments for security purposes. By using obstacle avoidance technology, Mobile robots can navigate through buildings and other structures to gather surveillance data without running into obstacles.

- **Maintenance and Inspection:**

Mobile robots can be used to inspect and maintain indoor facilities, such as pipes and other infrastructure components. By using obstacle avoidance technology, Mobile robots can navigate through tight spaces and around obstacles to identify potential maintenance issues and make repairs.

- **Delivery and Transportation:**

Mobile robots can be used to transport goods and supplies within indoor environments, such as factories and warehouses. By using obstacle avoidance technology, Mobile robots can navigate through narrow corridors and around obstacles to deliver goods and supplies to their intended destination.

By leveraging cutting-edge technology, Mobile robots can navigate through even the most complex indoor environments with ease, providing valuable data and insights for a variety of industries.

13. CONCLUSION

In conclusion, the project successfully implemented obstacle avoidance based on machine learning using a robot equipped with a model trained for monocular depth estimation. The system performed well in a variety of environments, successfully navigating around obstacles and reaching its destination. The use of machine learning for obstacle avoidance in robotics has significant potential, and the monocular depth estimation model used in this project provides a powerful tool for accurately detecting obstacles in the robot's environment. The model's compatibility with a wide range of hardware platforms makes it a highly scalable solution that can be easily adapted for use in different scenarios and applications. However, the project also highlights the need for large amounts of training data to ensure accurate predictions and the potential limitations of using ML for obstacle avoidance, such as the inability to detect certain types of obstacles.

Overall, the project provides a framework for future research in this area, and with further optimization and incorporation of other sensing techniques, the system could be improved to handle even more challenging environments and obstacles.

14. FUTURE PLAN

An outline of our future plan for this project is as follows:

1. Improve the accuracy of the depth estimation model by
 - Experimenting with different architectures such as ResNet, ENet to see if there are improvements in accuracy and efficiency.
 - Training the model with more diverse datasets to enhance its ability to generalize and handle various scenarios.
2. Develop a more efficient obstacle avoidance algorithm by
 - Experimenting with other algorithms such as Simultaneous Localization and Mapping (SLAM) and Rapidly exploring Random Trees (RRT) to see if there are improvements in performance.
 - Investigating the use of additional sensors such as LIDAR or RADAR to enhance the robot's ability to perceive its environment.
3. Incorporate learning-based approaches for the obstacle avoidance by
 - Using Reinforcement learning to enable the robot to learn from its experiences and adapt to changing environments.
 - Investigating the use of Imitation learning, where the robot learns from expert demonstrations, to improve its decision-making abilities.
4. Explore potential applications of the technology by
 - Investigating the use of the system in various applications such as search and rescue, surveillance, and inspection.

REFERENCES

- [1] R. Farkh, K. Al jaloud, S. Alhuwaimel, M. Tabrez Quasim and M. Ksouri, "A deep learning approach for the mobile-robot motion control system," *Intelligent Automation & Soft Computing*, vol. 29, no.2, pp. 423–435, 2021.
- [2] Y. Ma, Z. Wang, H. Yang and L. Yang, "Artificial intelligence applications in the development of autonomous vehicles: a survey," in *IEEE/CAA Journal of Automatica Sinica*, vol. 7, no. 2, pp. 315-329, March 2020.
- [3] Shumin Feng, Bijo Sebastian, Pinhas Ben-Tzvi, "A Collision Avoidance Method Based on Deep Reinforcement Learning" *Robotics* 2021, 10, 73.
- [4] Liyong Ma, Wei Xie, Haibin Huang, "Convolutional neural network based obstacle detection for unmanned surface vehicle", *Mathematical Biosciences and Engineering*, Volume 17, Issue 1, 845–861.
- [5] Karoline Kamil A. Farag, Hussein Hamdy Shehata, Hesham M. El-Batsh, "Mobile Robot Obstacle Avoidance Based on Neural Network with a Standardization Technique", *Journal of Robotics*, vol. 2021, 14 pages, 2021.
- [6] M. Z. Haque Zim, M. Sazzad Sarkar, Z. F. Nisi and N. C. Das, "LFNNR: Light Follower Neural Network Robot Conducted by Machine Learning Technique," 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus), 2021, pp. 1149-1154.
- [7] M. S. H. Sunny, E. Hossain, T. N. Mimma and S. Hossain, "An autonomous robot: Using ANN to navigate in a static path," 2017 4th International Conference on Advances in Electrical Engineering (ICAEE), 2017, pp. 291-296.
- [8] A. Medina-Santiago, J.L. Camas-Anzueto, J.A. Vazquez-Feijoo, H.R. Hernández-de León, R. Mota-Grajales, "Neural Control System in Obstacle Avoidance in Mobile Robots Using Ultrasonic Sensors", *Journal of Applied Research and Technology*, Volume 12, Issue 1, 2014, Pages 104-110.
- [9] K. Chi and M. R. Lee, "Obstacle avoidance in mobile robot using Neural Network," 2011 International Conference on Consumer Electronics, Communications and Networks (CECNet), 2011, pp. 5082-5085.
- [10] Ng, Sing Yee and Nur Syazreen Ahmad, "A Bug-Inspired Algorithm for Obstacle Avoidance of a Nonholonomic Wheeled Mobile Robot with Constraints," *Advances in Intelligent Systems and Computing*, 2019.
- [11] Zhao, C., Sun, Q., Zhang, C. et al, "Monocular depth estimation based on deep learning: An overview," *Sci. China Technol. Sci.* 63, 1612–1627 (2020).
- [12] Alhashim, Ibraheem and Peter Wonka, "High Quality Monocular Depth Estimation via Transfer Learning," *ArXiv abs/1812.11941* (2018): n. pag.

ACKNOWLEDGEMENT

Every piece of work is a result of tremendous result of many hands working behind the scene. To begin with, we would like to articulate this project as small journey which was a remarkable learning experience for us. Our group would like to thank everyone who assisted us in compiling this project titled “Obstacle Avoidance based on Machine Learning”.







We take this opportunity to express our profound gratitude and deep regards to our external guide Shri. Gaurav Baluni, Scientific Officer in Security Electronics & Cyber Technology Division (SECTD), BARC for his exemplary guidance, monitoring, cordial support and constant encouragement throughout the course of this project. It was a great opportunity and privilege to work under his able guidance, His sound knowledge and expertise helped us to complete this project smoothly without facing much difficulties.

We would also like to express our special thanks of gratitude to our internal project guide, Dr. T.P. Vinutha, internal project co-guide Prof. Amisha Bhoir as well as our principal, Dr. Bhavesh Patel who gave us the excellent opportunity to do this wonderful project. The successful completion of this project was only because of the extraordinary support, guidance, counseling and motivation.

Document Information

Analyzed document	GROUP_1_BE PROJECT REPORT_FINAL.pdf (D165318119)
Submitted	4/28/2023 8:23:00 AM
Submitted by	
Submitter email	aarushi.nandoskar_19@sakec.ac.in
Similarity	6%
Analysis address	rohan.borgalli.sakec@analysis.arkund.com

Sources included in the report

W	URL: https://www.passeidireto.com/arquivo/92309984/introduction-to-autonomous-mobile-robots/13 Fetched: 6/30/2021 3:26:38 PM		3
SA	EE50237 Project Report.pdf Document EE50237 Project Report.pdf (D124489891)		2
SA	Mazen Abed(229258909).pdf Document Mazen Abed(229258909).pdf (D155272860)		3
SA	ros_accessed_report.pdf Document ros_accessed_report.pdf (D124484426)		5
SA	ROSProjectReport.pdf Document ROSProjectReport.pdf (D112055885)		2
W	URL: https://stackoverflow.com/questions/44874681/features-matching-opencv Fetched: 5/7/2021 9:47:19 PM		2

Entire Document

i OBSTACLE AVOIDANCE BASED ON MACHINE LEARNING Submitted in partial fulfilment of the requirements of the degree of Bachelor of Engineering By Shrey Gupta BE – 7, 13 Aarushi Nandoskar BE – 7, 31 Narendra Parihar BE – 7, 34 Surya Prakash Yadav BE – 7, 60 Under the Guidance of Internal Guide: External Guide: Dr. T. P. Vinutha Shri. Gaurav Baluni Internal Co - Guide: Prof. Amisha Bhoir DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATION ENGINEERING SHAH AND ANCHOR KUTCHHI ENGINEERING COLLEGE MUMBAI UNIVERSITY 2022 – 2023

ii www.shahandanchor.com Tel: 022 2558 0854 Mahavir Education Trust's

SHAH & ANCHOR KUTCHHI ENGINEERING COLLEGE Mahavir Education Trust Chowk, W.T. Patil Marg, Chembur, Mumbai 400 088

Affiliated to University of Mumbai, Approved by D.T.E. & A.I.C.T.E. Certificate This is to certify that the report of the project entitled OBSTACLE AVOIDANCE BASED ON MACHINE LEARNING is a bonafide work of Shrey Gupta BE – 7, 13 Aarushi Nandoskar BE – 7, 31 Narendra Parihar BE – 7, 34 Surya Prakash Yadav BE – 7, 60 submitted to the UNIVERSITY OF MUMBAI during semester VII in partial fulfilment of the requirement for the award of the degree of BACHELOR OF ENGINEERING in ELECTRONICS AND TELECOMMUNICATION ENGINEERING _____

Dr. T. P. Vinutha Internal Guide Prof. Amisha Bhoir Internal Co-Guide

Dr. T. P. Vinutha Dr. Bhavesh Patel I/c Head of Department

Principal ISO 9001:2015 Certified