WESTERN UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL &
COMPUTER ENGINEERING

ECE 9000: M.ENG. PROJECT

Project Report

Title: Study and simulation of SLAM based robot navigation reinforced with OpenCV

SATHYANATH NANDAKUMAR

251085351

ACKNOWLEDGEMENTS

This project would not have been possible without the support of so many people. I would like to thank my advisor, Dr Jagath Samarabandu, Western University for his guidance and support throughout the course of my project work. For assistance and inputs regarding the SLAM algorithm I would like to thank Dr Ilia Polushin, Western University. For his invaluable assistance in helping me to understand and execute ROS I would like to thank Prof. Anis Koubaa, Prince Sultan University. I would like to thank Mohammad Navid Fekri, PhD Student at Western University for his invaluable input in executing SLAM using Xbox Kinect. Further I would like to thank the dev team at ROS, striving to make the platform as user friendly as possible. On a personal note, I would like to thank my housemates Thomas Joe and Bharat Raj for their constant support and encouragement in finishing the project. I would like to thank my girlfriend Anithya Balachandran for all her support and for having tolerated my horrible mood swings as I strived to finish this project. Finally, I wish to thank my parents for their endless encouragement and their support.

TABLE OF CONTENTS

INTRODUCTION

ABSTRACT

The future of robotics is in creating self-learning, human friendly robots. Merely the execution of a function such as navigation alone is not enough, rather, more crucial is what the robot can learn about itself and the environment it is in.

On that note, in this project, the goal is the reinforcement of the SLAM algorithm by means of object detection. Essentially given an indoor robot, granting it the ability to recognize the room it is in as it creates a map for SLAM navigation. The idea itself was presented in a paper that I had come across [1] and this project serves to show my own implementation of the same.

SCOPE

The scope of the project was taken to be the study and simulation of SLAM and robot navigation using Turtlebot3 along with object detection using OpenCV2, all on the ROS framework. The goal was the learn and understand the workings of software crucial to robot navigation

Completed Scope

- I have completed in entirety the study and simulation of ROS navigation and SLAM mapping
- In terms of computer vision with OpenCV and ROS, I have been able to execute a ball tracking application on the aforementioned platforms which has served to apply all that I have learnt in terms of processing using OpenCV 2. In the future, I will continue to work on executing an object detection application using the same platforms.

Future Scope

- The next stage is the real-life execution of the project for which significant investment is needed in terms of a robot base.
- This will also include interfacing a Xbox 360 Kinect camera and using its point-cloud output to create a map of the environment as shown in the simulation.
- The end goal is to have a robot that can recognize the room it navigates into on the basis of the objects identified in the room.

REPORT FLOW

- This report is broken down into three parts, namely navigation, SLAM mapping and Object detection.
- The report will introduce theory essential to the report followed by details of the equipment used to execute the same.
- The report will than deal with the study of SLAM and its simulation using turtlebot3 on Gazebo and Rviz to create a SLAM map.
- The report will delve into the basic navigation of a robot base using turtlesim and then its simulation using the previously defined SLAM map with turtlebot3 on Gazebo and Rviz.
- Finally, the report will speak of the computer vision part of the project namely the execution of a ball tracking application using OpenCV 2, ROS and a USB webcam.

ESSENTIAL THEORY

ROS

ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction and message-passing between processes. ROS is a distributed framework of processes (aka Nodes) that enables executables to be individually designed and loosely coupled at runtime. Given a project such as a drone used to capture images [2]:

- One node could be the camera capture and image processing
- Another node could be for the movement and control of the drone.
- ROS provides us with the means to connect the two and allows easy communication

ROS functions using two solutions namely topics and services:

- Nodes can publish on topics that other nodes can then subscribe to. For instance, in my code, a position node will publish the position of the robot that will be subscribed to by the velocity node for the robot.
- Services are a form of to and fro communication with one message type for request and one message type for response. The service consists of a request sent by the client and a response sent by the server.

SLAM

- SLAM (simultaneous localization and mapping) systems determine the orientation and position of a robot by creating a map of their environment while simultaneously tracking where the robot is within that environment. The most common SLAM systems rely on optical sensors, the top two being visual SLAM (VSLAM, based on a camera) or LiDAR-based (Light Detection and Ranging), using 2D or 3D LiDAR scanners. [3]
- Slam_gmapping: The gmapping package provides laser-based SLAM (Simultaneous Localization and Mapping), as a ROS node called slam_gmapping. [4]

CVBridge

- ROS passes around images in its own sensor_msgs/Image message format, but many users will want to use images in conjunction with OpenCV. CvBridge is a ROS library that provides an interface between ROS and OpenCV. CvBridge can be found in the cv_bridge package in the vision_opencv stack [5].

Gazebo and Rviz

- Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. To achieve ROS integration with stand-alone Gazebo, a set of ROS packages named gazebo_ros_pkgs provides wrappers around the stand-alone Gazebo.[6]
- Rviz is a 3d visualization tool for ROS applications. It provides a view of your robot model, capture sensor information from robot sensors, and replay captured data. It can display data from camera, lasers, from 3D and 2D devices including pictures and point clouds. [7]

EQUIPMENT SPECIFICATIONS

The project has been executed using the following:

- Framework: ROS Melodic Morenia running on Ubuntu 18.04
- Technical computing: MATLAB 2019
- Computer Vision: OpenCV 2, CVbridge for ROS
- Simulation: Gazebo version 9.0.0, Rviz 1.13.13 (melodic).
- Webcam: Logitech C270 HD
- IDE: Microsoft Visual Studio

SLAM

STUDY ON THE SLAM ALGORITHM

Execution and comparison of two filters commonly used in autonomous robotics namely EKF and UKF filters, these algorithms act as the basis for further topics such as EKF-SLAM.

EXTENDED KALMAN FILTER

<u>Problem Statement:</u>

To study and execute the Extended Kalman Filter for a given non-linear system.

- The Extended Kalman filter is to be executed to estimate a sine wave.
- The position estimate is to be plotted against the true position and measurements over time.
- The frequency estimate is to be plotted against the true frequency and measurements over time.

<u>Algorithm [8]:</u>

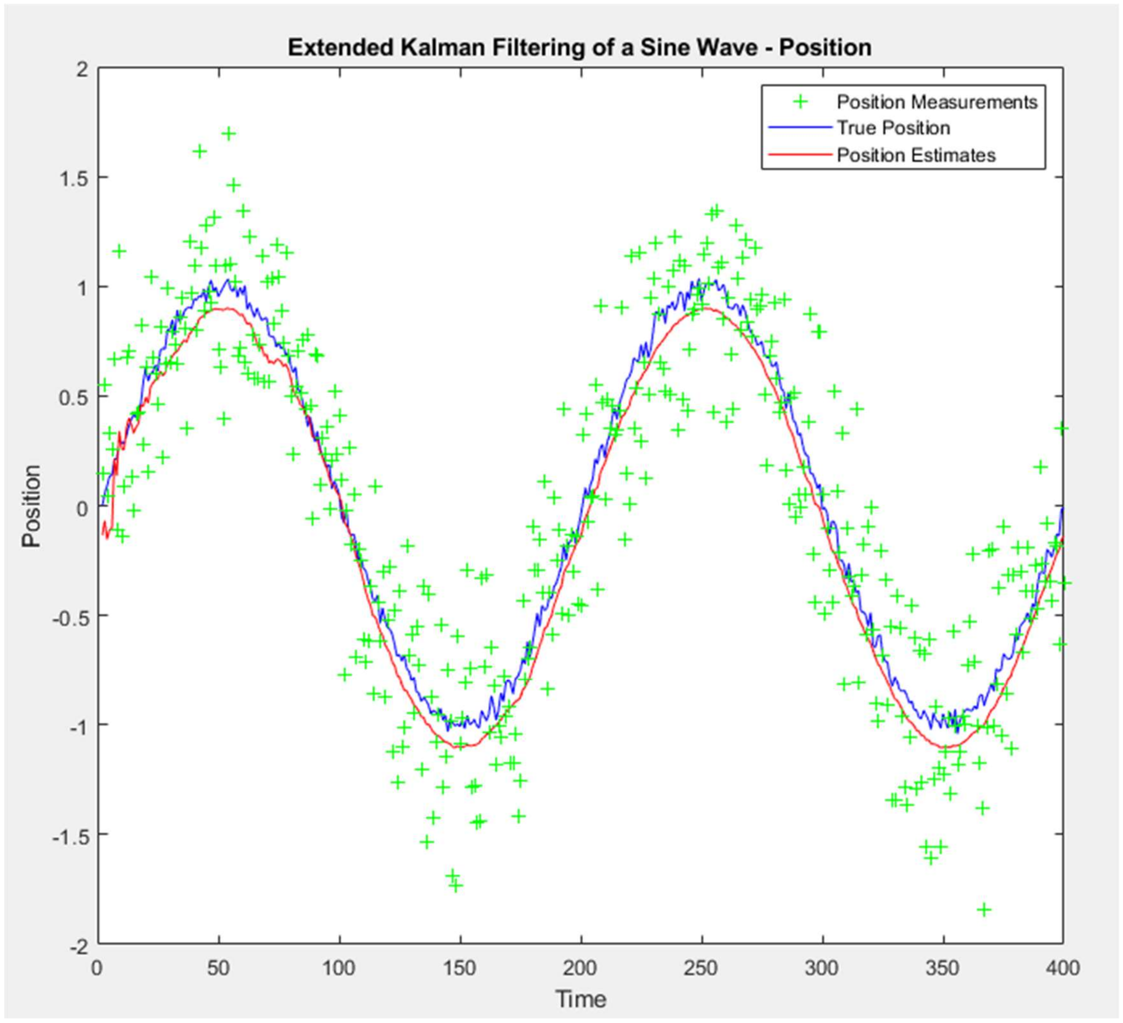
```
1. Algorithm Extended_Kalman_Filter $(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$:

2.     $\bar{\mu}_t = g(u_t, \mu_{t-1})$
3.     $\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$

4.     $K_t = \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + Q_t)^{-1}$
5.     $\mu_t = \bar{\mu}_t + K_t (z_t - h(\bar{\mu}_t))$
6.     $\Sigma_t = (I - K_t H_t) \bar{\Sigma}_t$

7.     return $\mu_t$, $\Sigma_t$
```

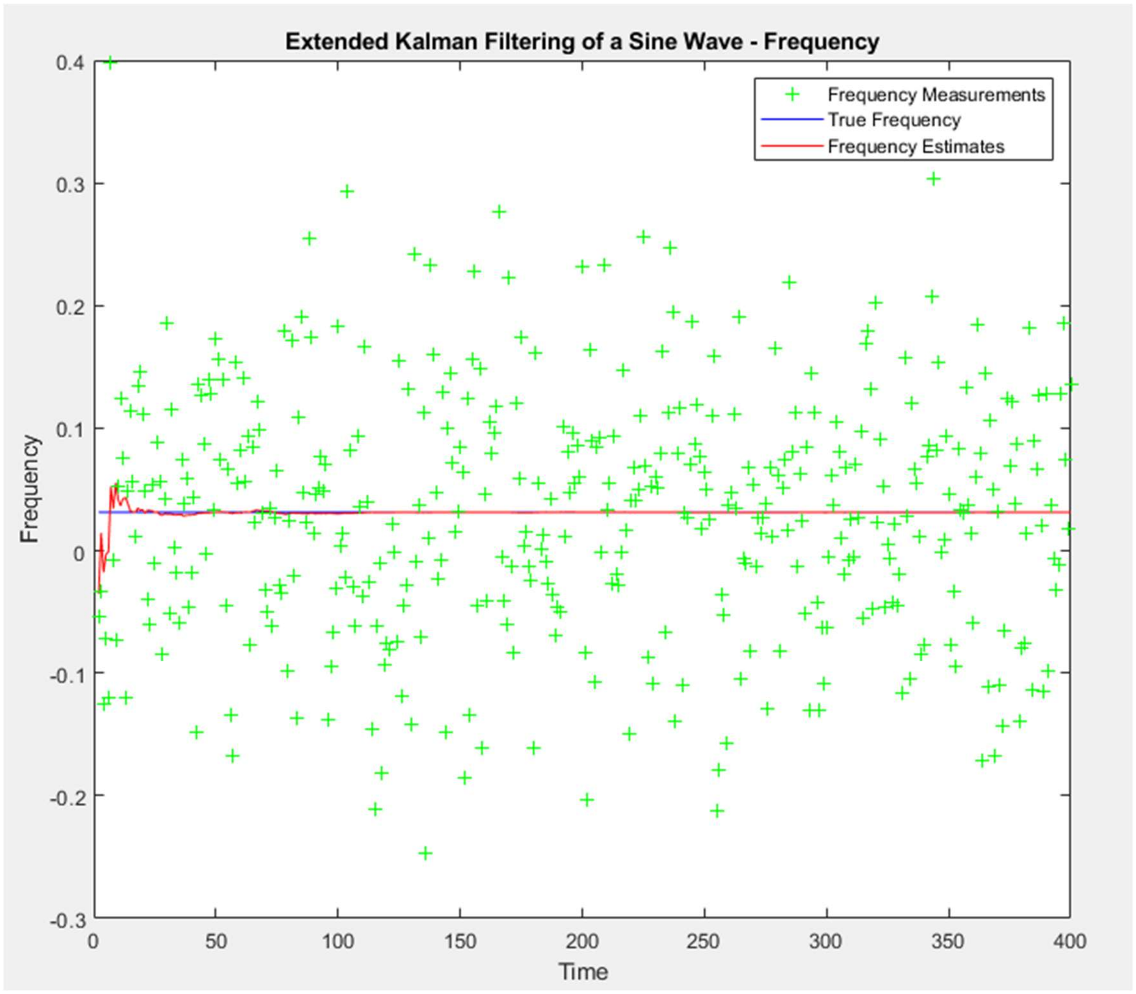
- In step 2: The linear state prediction $\bar{\mu}_t = A_t\mu_{t-1} + B_t\mu_t$ of the Kalman filter is replaced with the nonlinear state prediction $\bar{\mu}_t = g(\mu_t, \mu_{t-1})$ in EKF.
- In step 3: In EKF, the Jacobian $G_t = \delta g(\mu_t, x_{t-1}) / \delta x_{t-1}$ replaces the linear system matrix At used in Kalman filter.
- In step 5: The linear measurement prediction $C_t\bar{\mu}_t$ of the Kalman filter is replaced with the nonlinear measurement prediction $h(\bar{\mu}_t)$ in EKF.
- In steps 4 & 6: The linear system matrix C_t used in Kalman filter is replaced in EKF with the Jacobian $H_t := \delta h(x_t) / \delta x_t$.

System Description

- True state: $x_t = \text{Sin}(t) + \text{noise}$
- Measurement: $z_t = x_t + \text{noise}$
- True initial state: x = [0;
  3*pi/300];
- Initial aposteriori state estimate: μ = [1;
  1*pi/300];
- Process noise covariance: R = [0.001 0;
  0 0];
- Measurement noise covariance: Q = [0.1 0;
  0 0.01];
- Initial aposteriori error covariance estimate: Σ = [1 0;
  0 1];

Output:





<u>Observation:</u>

- The main strengths of EKF is its simplicity and computational efficiency.
- The main feature (also a limitation) of EKF is that it approximates state transition and measurement functions using Taylor expansions.
- The goodness of the approximation used in EKF depends on two factors:
  – the degree of uncertainty;
  – the degree of (local) nonlinearity of the approximated functions.

UNSCENTED KALMAN FILTER

Problem Statement:

To study and execute the Unscented Kalman Filter for a given non-linear system.

- The Unscented Kalman filter is to be executed to estimate a sine wave.
- The position estimate is to be plotted against the true position and measurements over time.
- The frequency estimate is to be plotted against the true frequency and measurements over time.

Algorithm: [9]

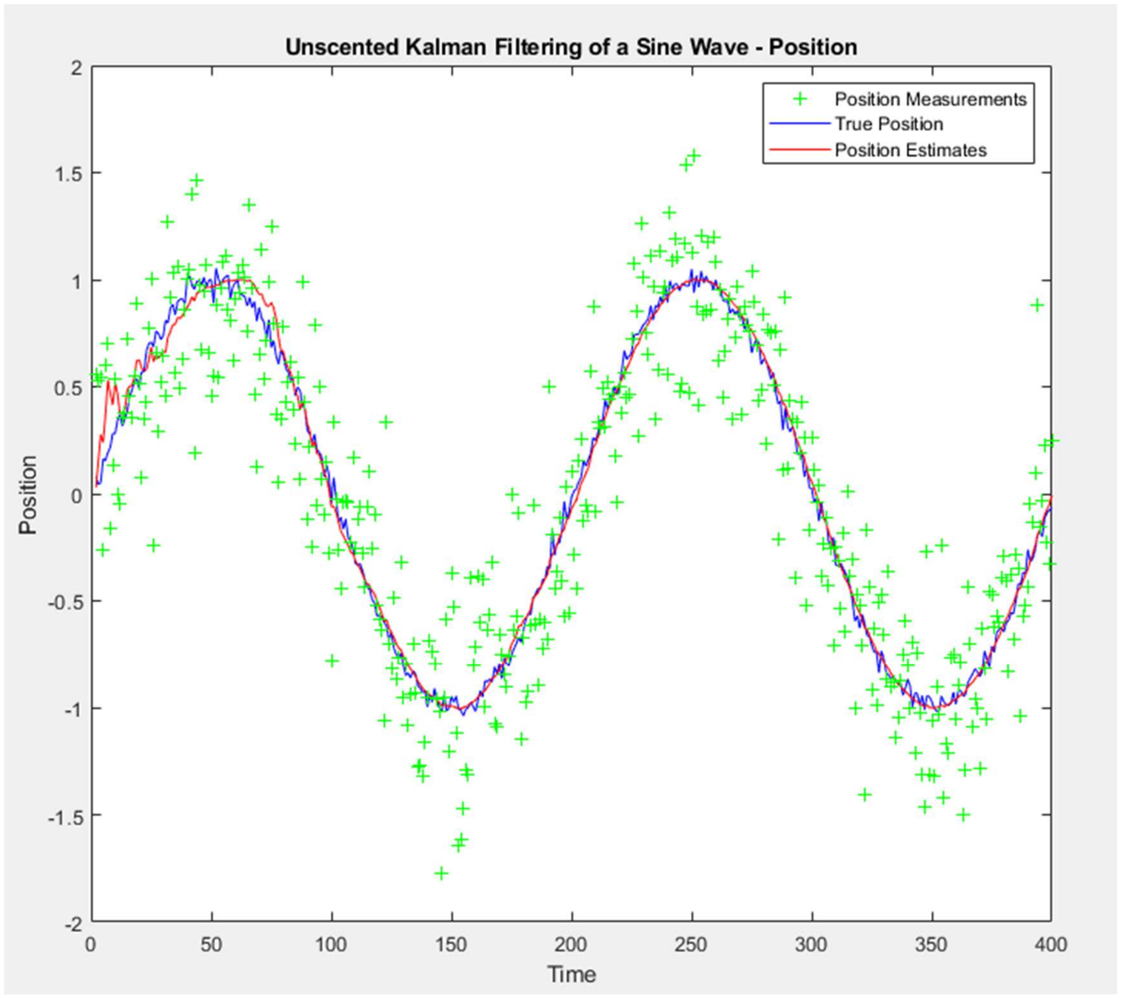
```
1.  **Algorithm Unscented_Kalman_Filter** $(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$:

2.      $\mathcal{X}_{t-1} = \left(\mu_{t-1} \quad \mu_{t-1} + \gamma\sqrt{\Sigma_{t-1}} \quad \mu_{t-1} - \gamma\sqrt{\Sigma_{t-1}}\right)$
3.      $\bar{\mathcal{X}}^*_{t-1} = g\left(u_t, \mathcal{X}_{t-1}\right)$
4.      $\bar{\mu}_t = \sum_{i=0}^{2n} w_m^{[i]}\, \bar{\mathcal{X}}_t^{*[i]}$
5.      $\bar{\Sigma}_t = \sum_{i=0}^{2n} w_c^{[i]} \left(\bar{\mathcal{X}}_t^{*[i]} - \bar{\mu}_t\right)\left(\bar{\mathcal{X}}_t^{*[i]} - \bar{\mu}_t\right)^T + R_t$
6.      $\bar{\mathcal{X}}_t = \left(\bar{\mu}_t \quad \bar{\mu}_t + \gamma\sqrt{\bar{\Sigma}_t} \quad \bar{\mu}_t - \gamma\sqrt{\bar{\Sigma}_t}\right)$
7.      $\bar{\mathcal{Z}}_t = h\left(\bar{\mathcal{X}}_t\right)$
8.      $\hat{z}_t = \sum_{i=0}^{2n} w_m^{[i]}\, \bar{\mathcal{Z}}_t^{[i]}$
9.      $S_t = \sum_{i=0}^{2n} w_c^{[i]} \left(\bar{\mathcal{Z}}_t^{[i]} - \bar{\mu}_t\right)\left(\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t\right)^T + Q_t$
10.     $\bar{\Sigma}_t^{x,z} = \sum_{i=0}^{2n} w_c^{[i]} \left(\bar{\mathcal{X}}_t^{[i]} - \bar{\mu}_t\right)\left(\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t\right)^T$
11.     $K_t = \bar{\Sigma}_t^{x,z} S_t^{-1}$
12.     $\mu_t = \bar{\mu}_t + K_t\,(z_t - \hat{z}_t)$
13.     $\Sigma_t = \bar{\Sigma}_t - K_t S_t K_t^T$

14.     **return** $\mu_t$, $\Sigma_t$
```

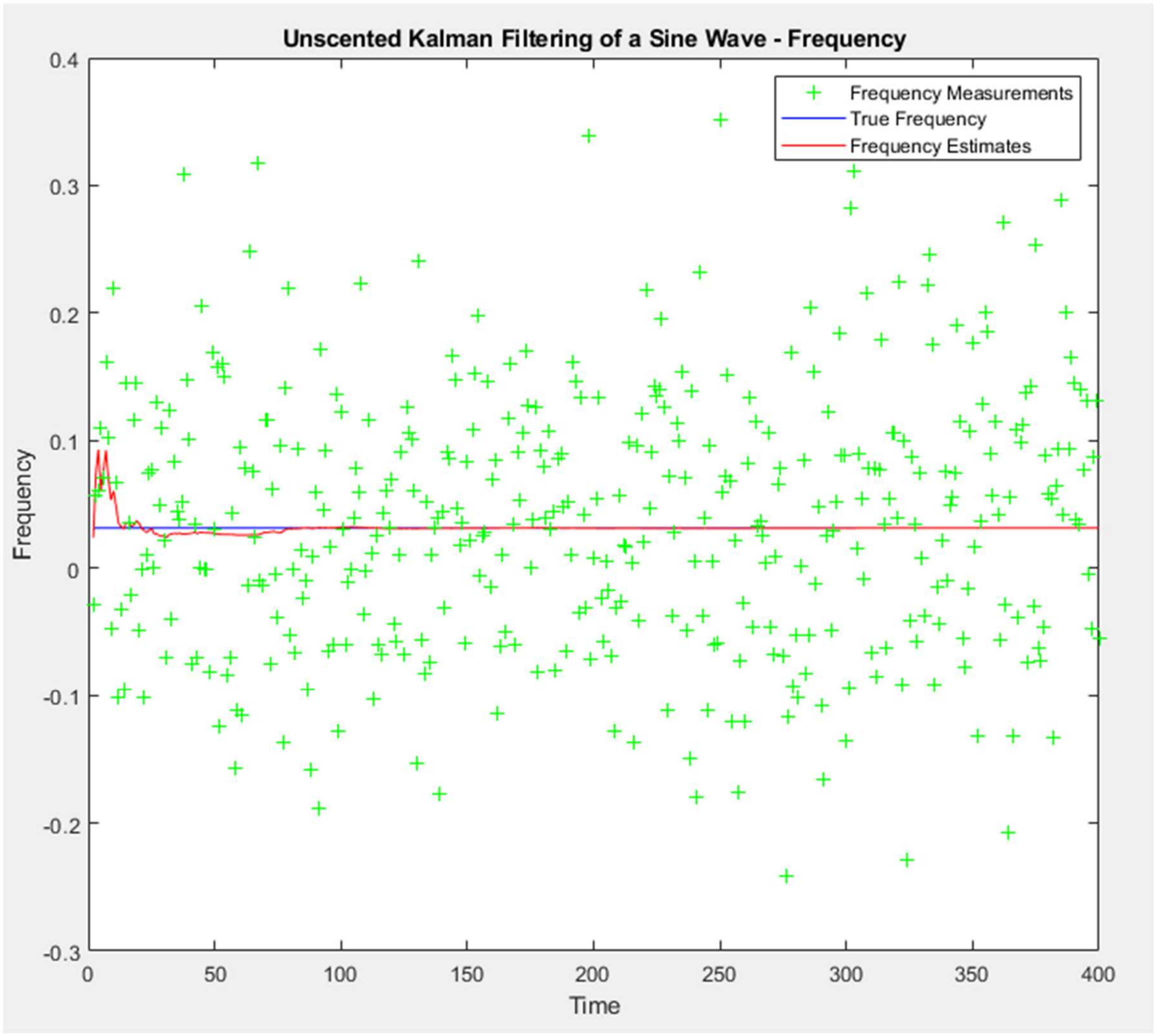
- The UKF algorithm is recursive. At each step t, it calculates the Gaussian belief μt, ^t based on the previous belief μ_t, Σ_t, control input u_t, and measurement z_t.
- The UKF algorithm consists of prediction (steps 2-5) and measurement update (steps 6-13).

System Description:

- True state: $x_t = \mathrm{Sin}(t) + \text{noise}$, Measurement: $z_t = x_t + \text{noise}$
- True initial state: x = [0;
  3*pi/300];
- Initial aposteriori state estimate: μ = [1;
  1*pi/300];
- Process noise covariance: R = [0.000   0;
  0   0.000];
- Measurement noise covariance: Q = [0.0001   0;
  0   0.001];
- Initial aposteriori error covariance estimate: Σ = [1   0;
  0   1];
- Dimension of the state vector: n = 2

Output:





<u>Observation:</u>

- The computational complexity of EKF and UKF is approximately the same. In practice, EKF is usually slightly faster than UKF.
- For nonlinear systems, UKF provides equally good or better estimates as compared to EKF.
- Improvement brought by UKF depends on specific nonlinearities and the prior state uncertainty.

TURTLEBOT3 SIMULATION OF LIDAR SLAM MAPPING

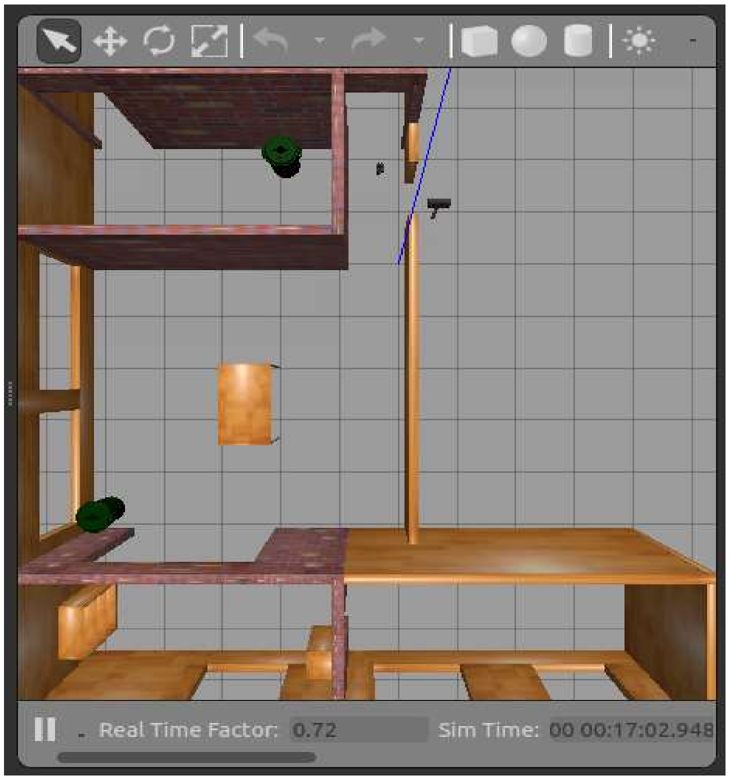
Problem Statement:

To map the simulation of a turtlebot3 in an indoor environment using LiDAR SLAM.

Methodology [10]:

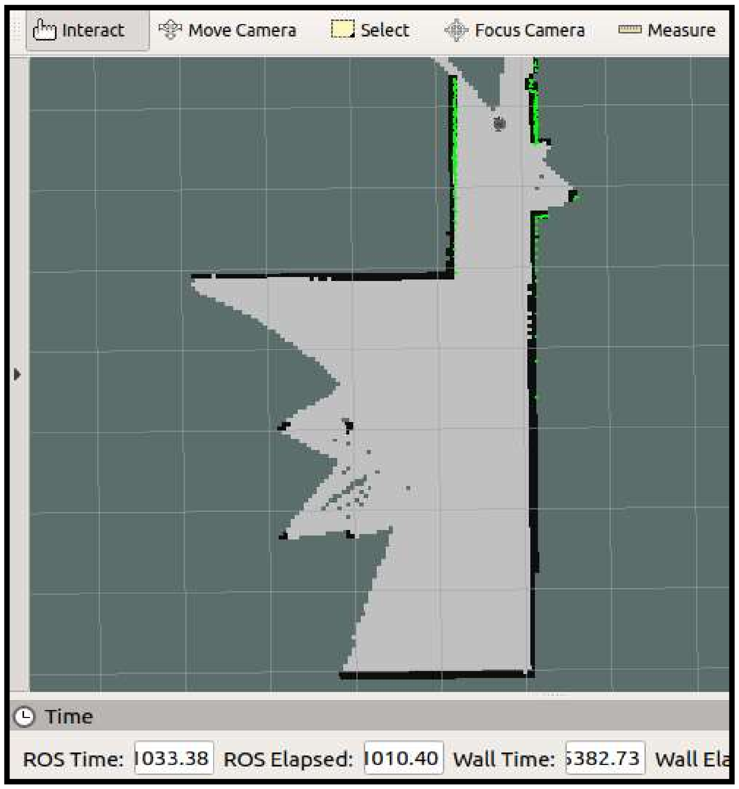
- Launch the simulation of a Burger turtlebot using the following command:
```
roslaunch turtlebot3_gazebo turtlebot3_house.launch
```
- Execute slam gmapping
```
roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```
- Control the simulated robot using the following command:
```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```
- Move the robot around the environment until a complete map has been created.

Output:



Real Time Factor: 0.72 Sim Time: 00 00:17:02.948

Fig 1. Simulated turtlebot3 in a house environment



ROS Time: 1033.38 ROS Elapsed: 1010.40 Wall Time: 5382.73 Wall Ela

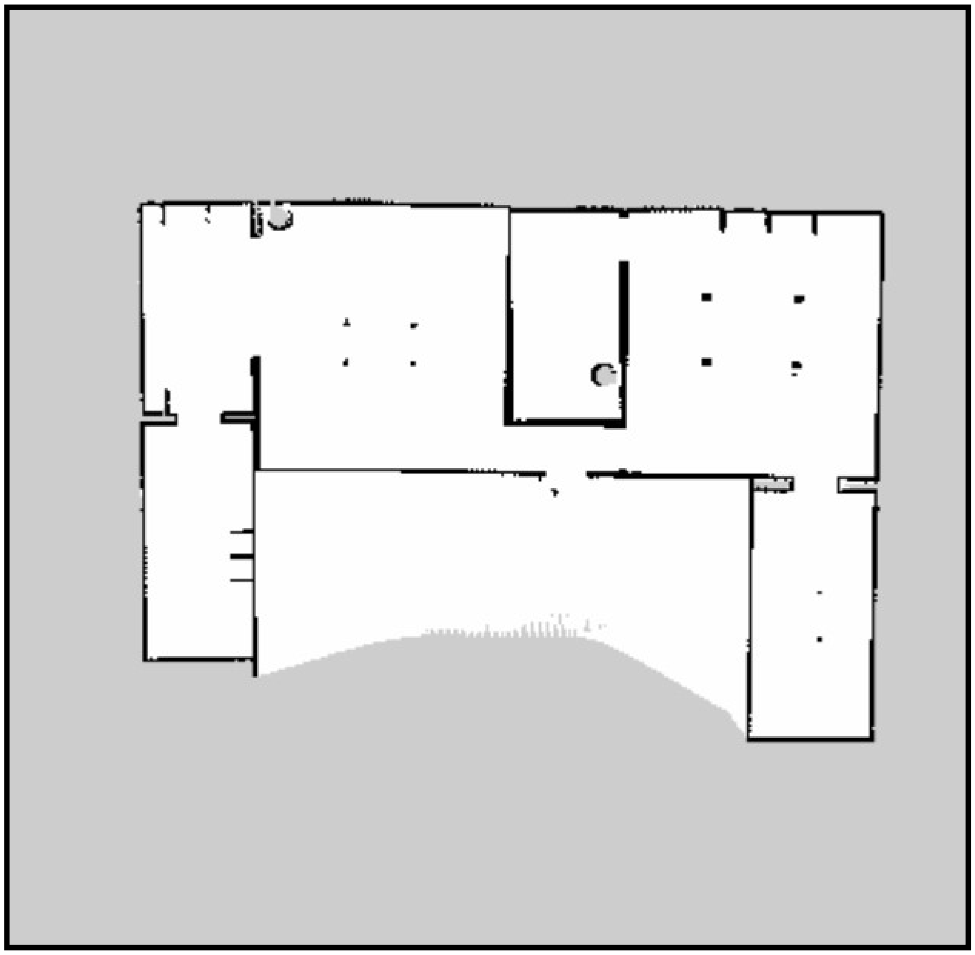
Fig 2. Moving the robot and mapping the environment

Upon saving the final map we receive two files, the yaml file which defines the parameters and the pgm file for the map which describing features of the created map as shown:

- h_map.yaml

```
image: /home/sathyanath/ROS/h_map.pgm
resolution: 0.050000
origin: [-10.000000, -10.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

- h_map.pgm :
    - The gray area indicates areas for which there isn’t any information available or there is insufficient information to map it.
    - The white areas are fully mapped out.
    - The black areas are the obstacles and walls of the indoor environment.



ROBOT NAVIGATION

BASIC NAVIGATION

Problem Statement:

The robot base must be able to move from the starting point to the goal location. The execution would involve continuously communicating the current location of the robot and simultaneously receiving velocity commands. Further, functions need to be defined for forward motion, rotational motion and goal seek.

Methodology:

- Ground up code has been created for simple navigation from one point to the next assuming no obstacles in the path of the robot in Python, simulating the behavior of a Roomba.
- The code is executed over the ROS framework using the concept of topics to continuously publish messages on the velocity topic and receive messages on the position topic.
- Turtlesim is used to simulate the behavior of the robot. This code will execute exactly the same way if executed for a simulation of real-life robots such as Turtlebot2 by changing the topic it is publishing to.
- Detailed information about setting up and executing ROS is available at [11]

Algorithm [12]:

Goal Seek function

- Receive the x and y goal coordinates and the tolerance.
- $V_f = \text{KV} \times \sqrt{(x_f - x_i)^2 + (y_f - y_i)^2}$, where V_f is the calculated linear velocity, x_f and y_f are the coordinates of the final position + tolerance, x_i and y_i are the current position values of the robot as it approaches the goal and KV is the proportional constant determined by trial and error.
- $A_f = \text{KH} \times \left[\tan^{-1}\frac{(y_f - y_i)}{(x_f - x_i)} - \theta\right]$, where A_f is the calculated angular velocity, x_f and y_f are the coordinates of the final position + tolerance , x_i and y_i are the current position values of the robot with θ being the current orientation as it approaches the goal, and KH is the proportional constant determined by trial and error.
- Publish the calculated linear velocity on x component of the linear velocity message and the calculated angular velocity on the z component of the angular velocity message.
- Stop publishing when the position is calculated to be at the goal location.

Linear motion function:

- Receive the speed, distance and direction of motion
- Publish the speed as the linear velocity on x component of the linear velocity message
- Calculated distance moved = Calculated distance moved + $\sqrt{(x_c - x_i)^2 + (y_c - y_i)^2}$, where x_c and y_c are the coordinates of the current position, x_i and y_i are the initial position values of the robot.
- Stop publishing when the calculated distance is greater or equal to the required distance.

Angular motion function:

- Receive the angular speed in degree, relative angle in degree and the direction of rotation
- Publish the angular speed as the angular velocity on z component of the angular velocity message.
- Current orientation = angular speed × $(t_c - t_i)$, where t_c is the current time and t_i is the initial time.
- Stop publishing when the calculated angle is greater or equal to the required angle

<u>Execution:</u>

- Launch rosmaster with a user defined launch file, opening turtlesim.:

```
<launch>
   <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
</launch>
```

- Execute the python file via the catkin workspace.

<u>Output:</u>

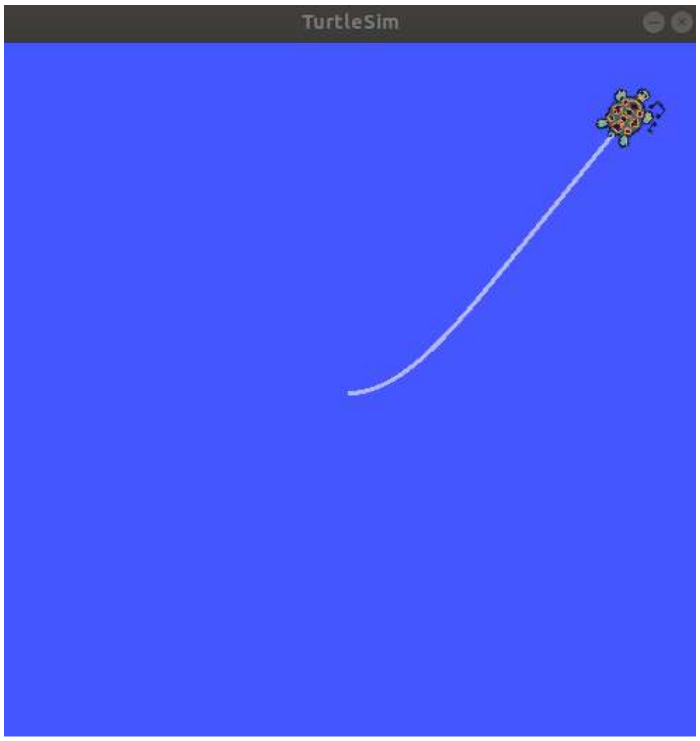


Fig 2. Turtlesim: Go to goal function

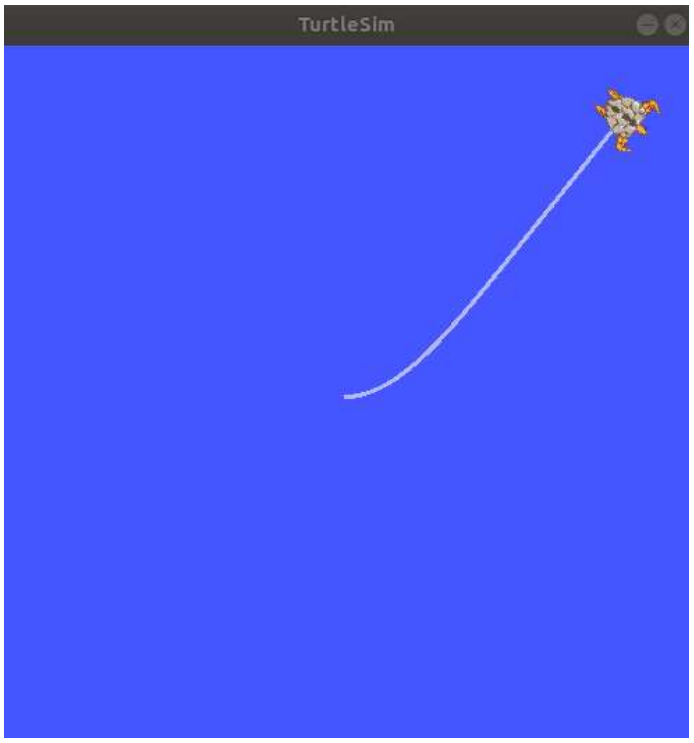
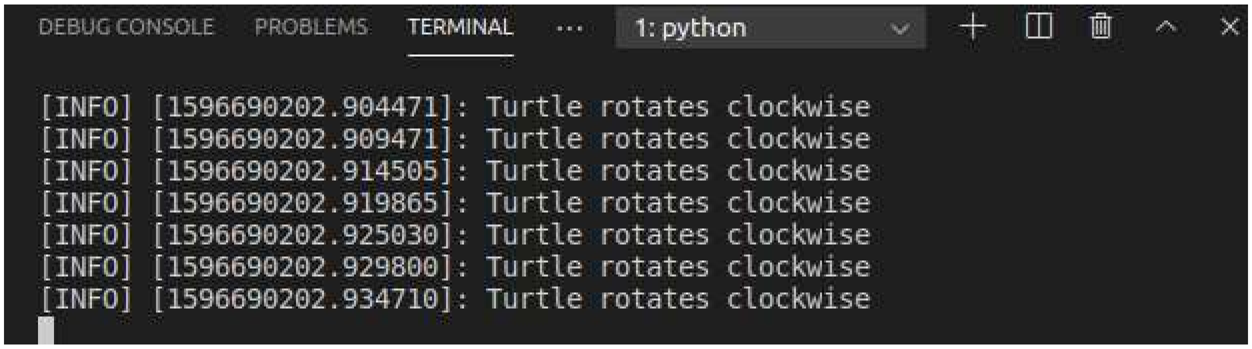
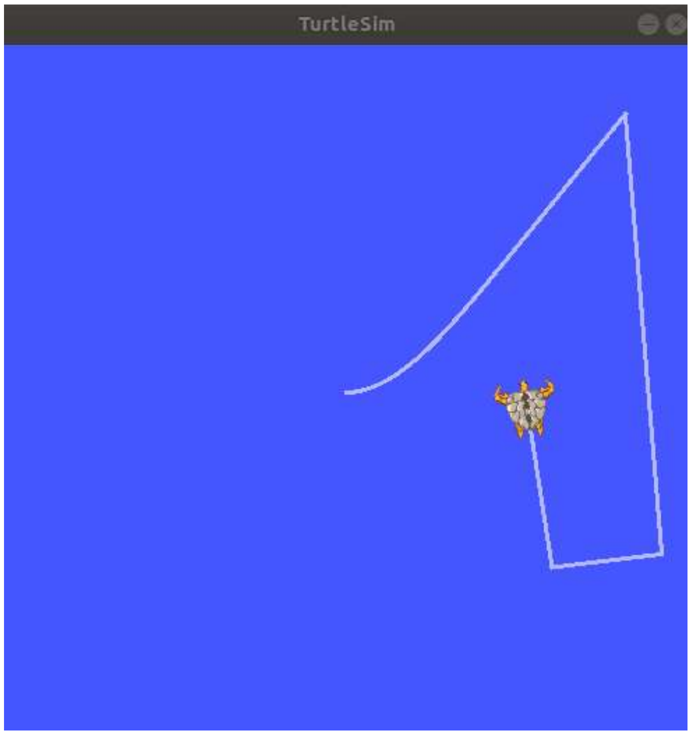


Fig 3. Turtlesim: Rotate function



```
[INFO] [1596690202.904471]: Turtle rotates clockwise
[INFO] [1596690202.909471]: Turtle rotates clockwise
[INFO] [1596690202.914505]: Turtle rotates clockwise
[INFO] [1596690202.919865]: Turtle rotates clockwise
[INFO] [1596690202.925030]: Turtle rotates clockwise
[INFO] [1596690202.929800]: Turtle rotates clockwise
[INFO] [1596690202.934710]: Turtle rotates clockwise
```

Fig 4. Rotate output window



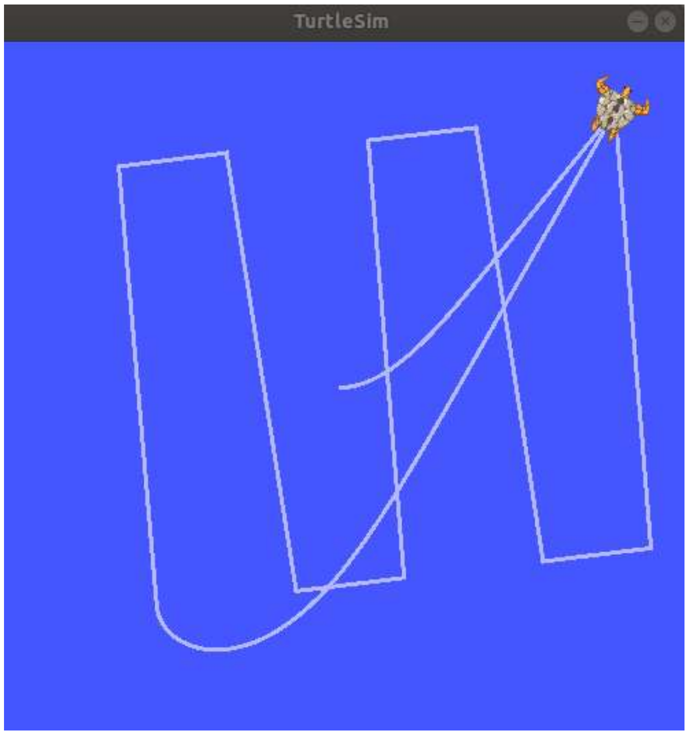
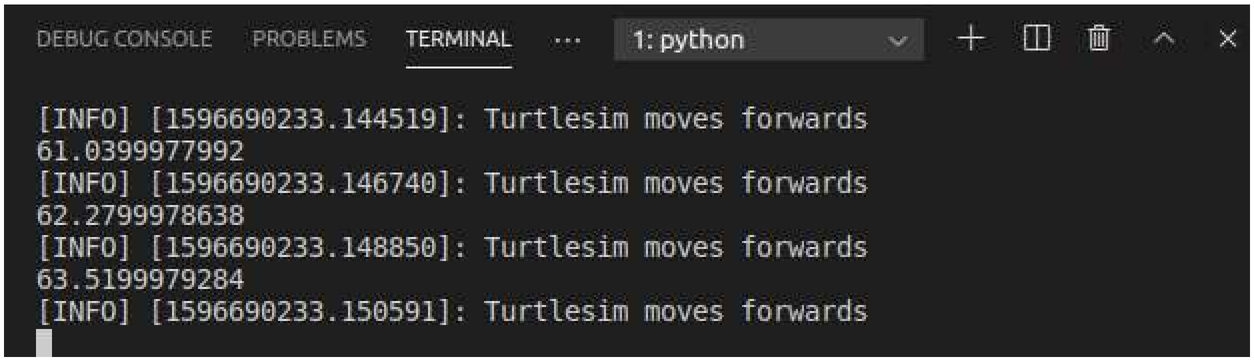


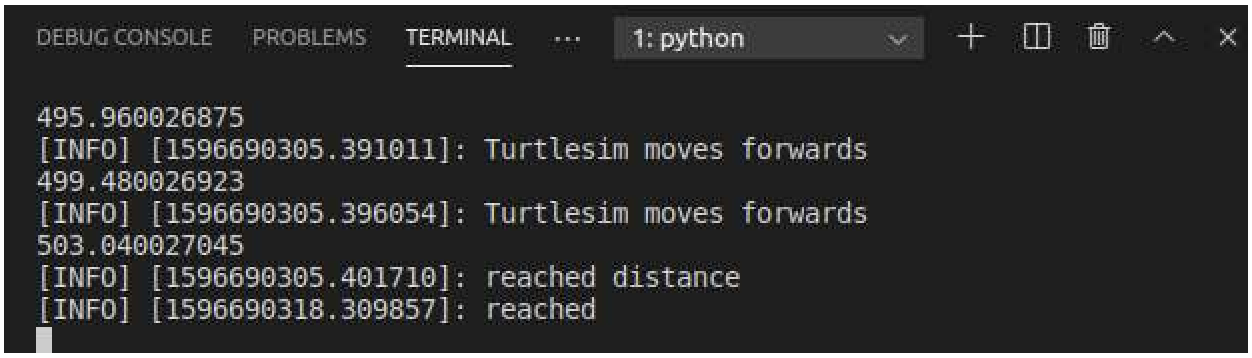
Fig 5. Turtlesim: Linear motion function

Fig 6. Turtlesim: Return to goal function



```
[INFO] [1596690233.144519]: Turtlesim moves forwards
61.0399977992
[INFO] [1596690233.146740]: Turtlesim moves forwards
62.2799978638
[INFO] [1596690233.148850]: Turtlesim moves forwards
63.5199979284
[INFO] [1596690233.150591]: Turtlesim moves forwards
```

Fig 7. Linear motion output window



```
495.960026875
[INFO] [1596690305.391011]: Turtlesim moves forwards
499.480026923
[INFO] [1596690305.396054]: Turtlesim moves forwards
503.040027045
[INFO] [1596690305.401710]: reached distance
[INFO] [1596690318.309857]: reached
```

Fig 8. Return to goal output window

TURTLEBOT3 SIMULATION OF NAVIGATION

Problem Statement:
To simulate the behavior of a robot and execute the goal seek function in an indoor environment with the previously defined SLAM map.

Methodology:

- Simulation of a Turtlebot3 Burger has been done using Gazebo and rviz showcasing the navigation using the previously defined SLAM map.
- The go to goal function will be executed on this setup.

Algorithm [13]:

- Receive the goal location in terms of x and y
- Execute the move_base navigation node using Actionlib
- Define endpoint as an object of MoveBaseGoal ( )
- Use endpoint to define the reference frame, goal positions and goal orientation.
- Use an object of actionlib named a1 to feed the endpoint into send_goal.
- Execute until the goal has been reached or the allotted time for execution has expired.

Execution:

- Launch the turtlebot3 simulator
  `roslaunch turtlebot3_gazebo turtlebot3_house.launch`
- Launch the created map file through rviz
  ```
  roslaunch turtlebot3_navigation turtlebot3_navigation.launch
  map_file:=/home/sathyanath/ROS/h_map.yaml
  ```
- Execute the goal seek python file via the catkin workspace.

Output:

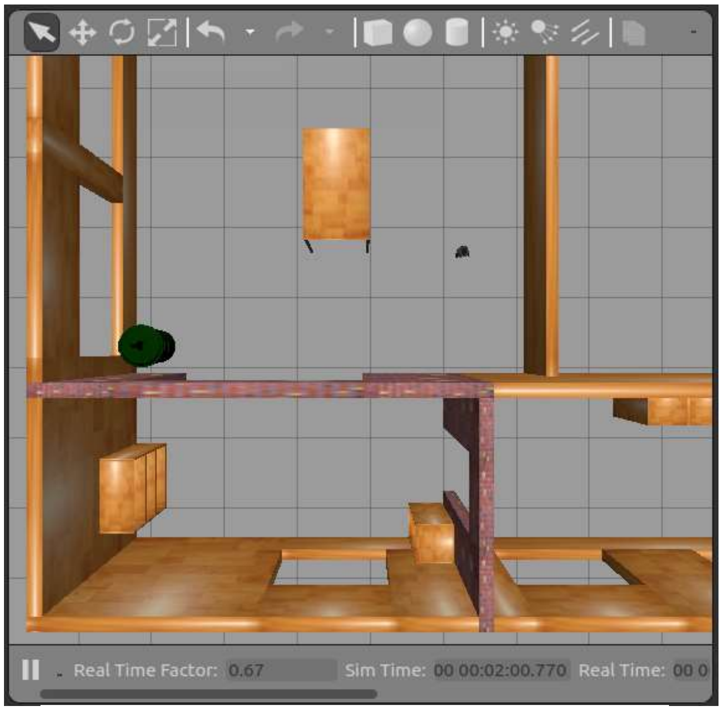


Fig 9. Indoor env simulation on Gazebo:
Determining a route to the destination

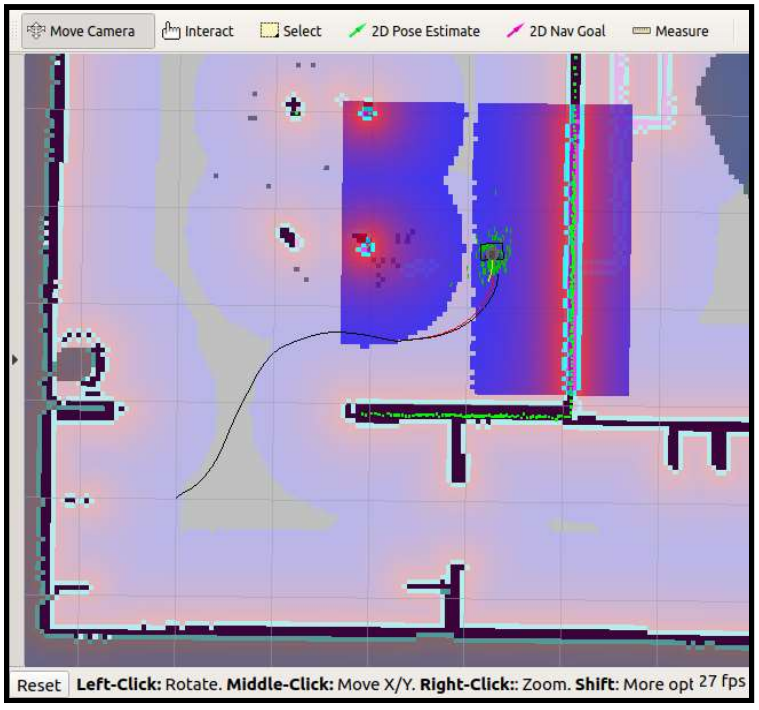

Move Camera
Interact
Select
2D Pose Estimate
2D Nav Goal
Measure
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Zoom. Shift: More opt
27 fps

Fig 10. Simulation of the defined SLAM map on Rviz:
Determining a route to the destination

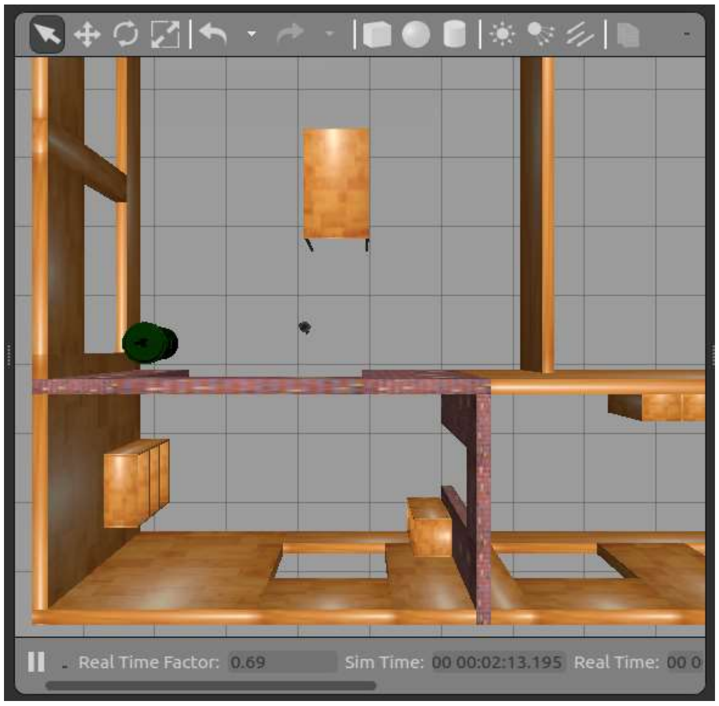


Fig 9. Indoor env simulation on Gazebo:
Traversing the route to the destination

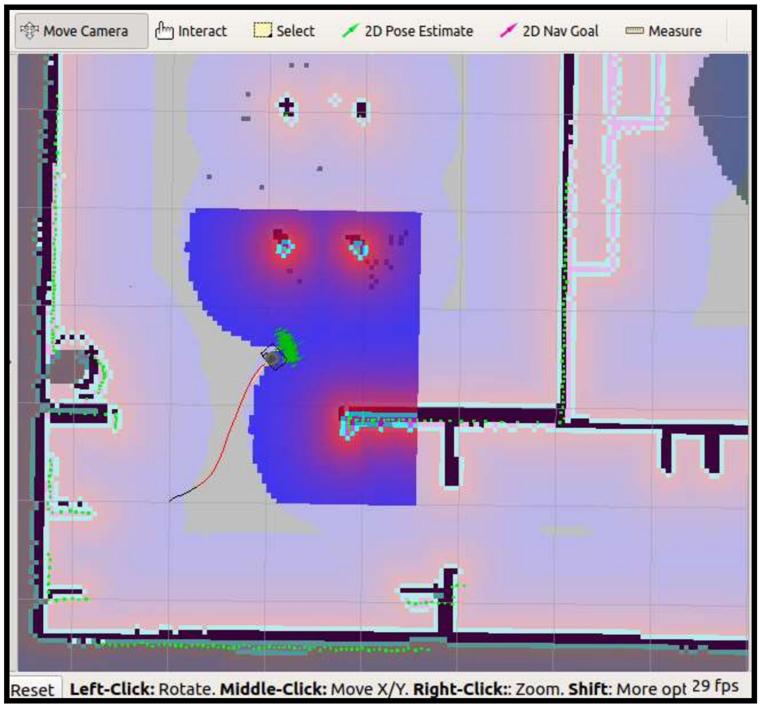

Move Camera
Interact
Select
2D Pose Estimate
2D Nav Goal
Measure
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Zoom. Shift: More opt
29 fps

Fig 10. Simulation of the defined SLAM map on Rviz:
Traversing the route to the destination

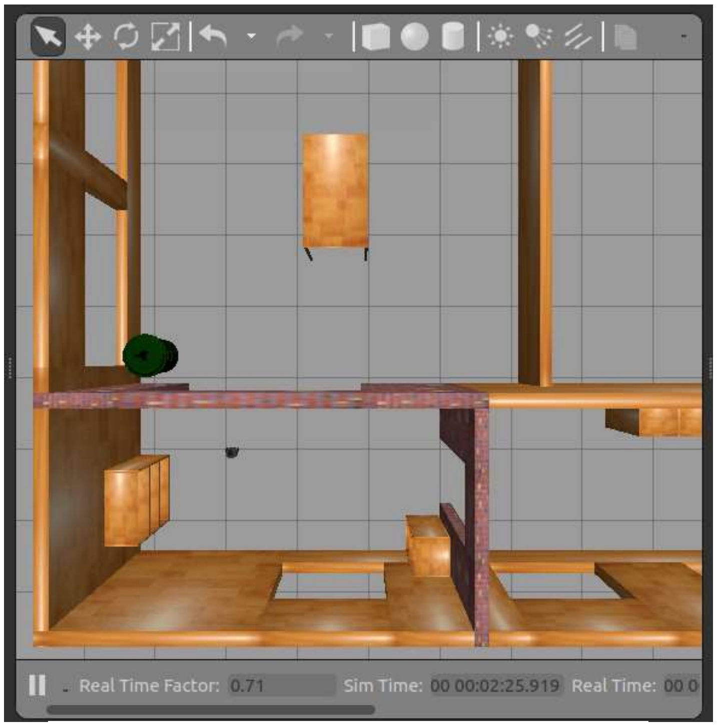


Fig 9. Indoor env simulation on Gazebo: Arrival at the destination

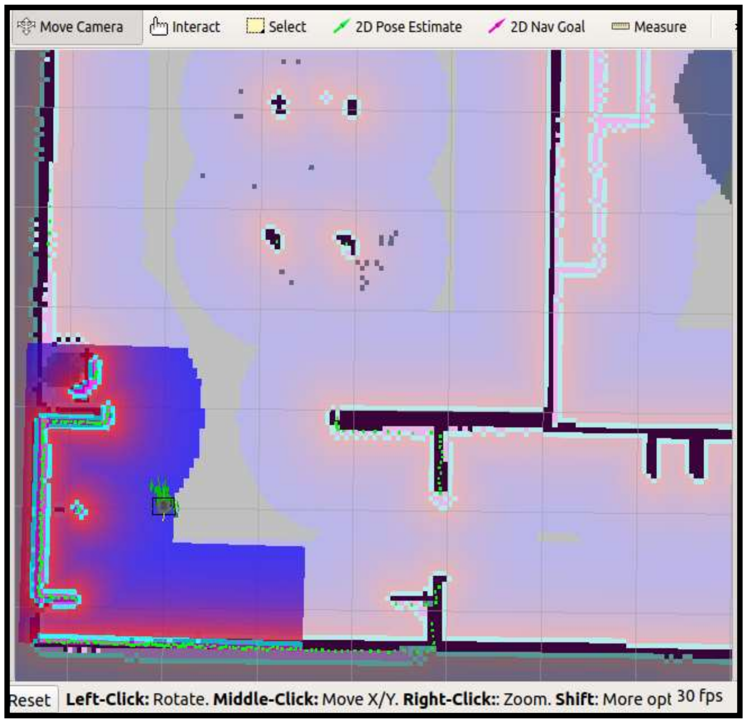
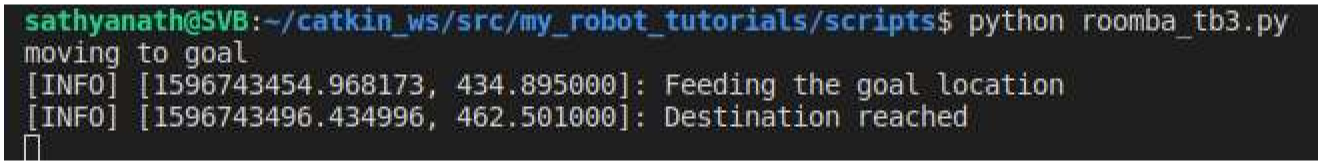

Move Camera
Interact
Select
2D Pose Estimate
2D Nav Goal
Measure
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Zoom. Shift: More opt
30 fps

Fig 10. Simulation of the defined SLAM map on Rviz: Arrival at the destination



```
sathyanath@SVB:~/catkin_ws/src/my_robot_tutorials/scripts$ python roomba_tb3.py
moving to goal
[INFO] [1596743454.968173, 434.895000]: Feeding the goal location
[INFO] [1596743496.434996, 462.501000]: Destination reached
```

Fig 11. Output window

COMPUTER VISION WITH OPENCV AND ROS

Problem Statement:

To detect and track a ball by processing the input via a USB camera, this includes reading the video file, converting it into a format suitable for processing, processing the video image by image and finally drawing a circle around the identified ball as it tracks across the screen.

Methodology:

- The video stream will be accessed through ROS to ensure the data can be easily accessible across other nodes.
- The stream will be converted into an OpenCV compatible form using CVBridge for ROS.
- The video stream will then be processed into giving the desired output using color filtering, contour detection and contour marking, using OpenCV 2.

Algorithm [13]:

Video input and conversion function

- Access the USB webcam input via ROS, note the format of the input.
- Use CVbridge to convert the video into an OpenCV compatible format

Color filtering function

- Convert the RGB video into an HSV video, to better account for the varying lighting conditions
- Assign upper and lower limits for the color mask
- Return a masked video

Contour detection function

- Detect the contours using cv2.findContours and return the required array of values
- Feed the original image, the masked image and the detected contours into the contour marking function

Counter marking function

- Iterate across the contour array and determine the contour center based on the area and perimeter of each contour.
- Draw circles, keeping each determined center point of the ball as the circle center.
- The area taken can be used to exclude noise from the processed image.

Execution:

- Run the USB webcam through ROS accessing the proper format
```
rosrun usb_cam usb_cam_node _pixel_format:=yuyv
```

- Execute the ball tracking python file

Output:

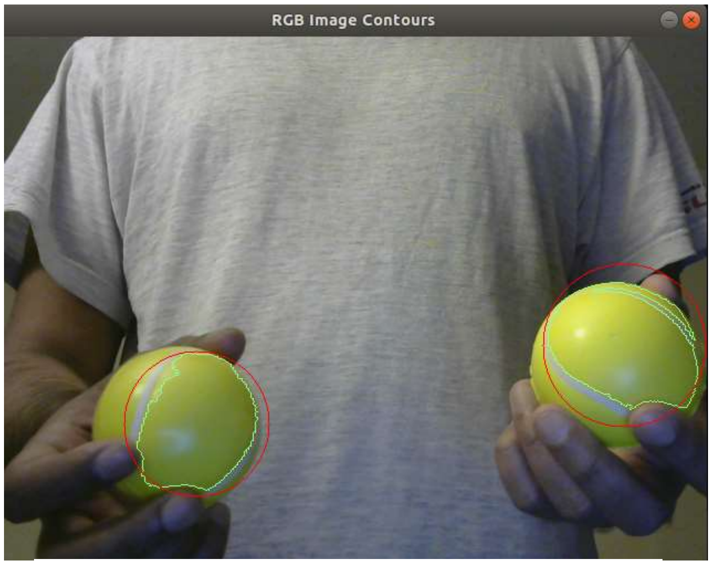


Fig 12. Original RGB video

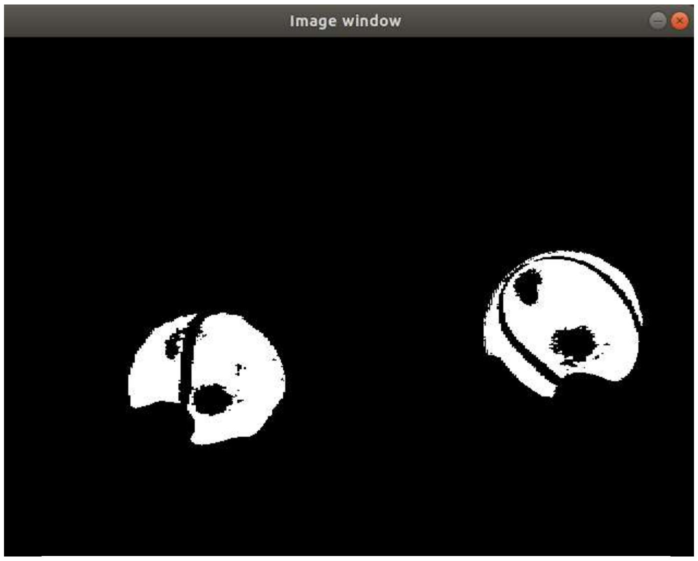


Fig 13. Masking of the RGB video

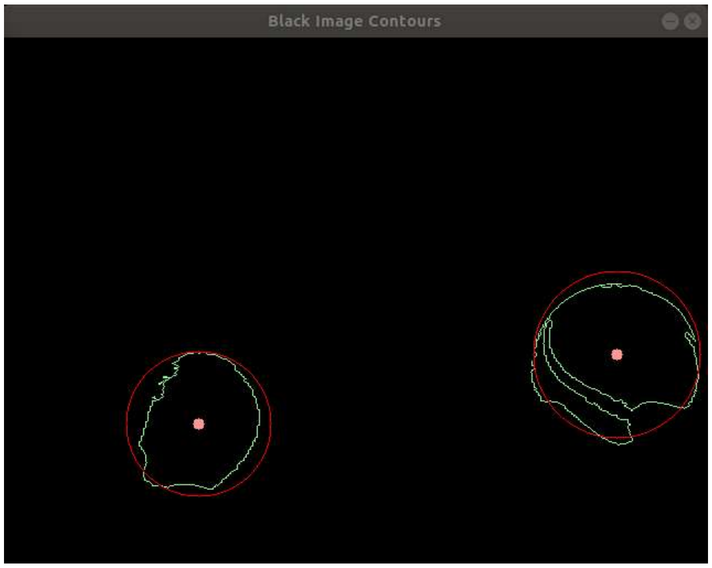


Fig 14. Contour detection and tracking

REFERENCES

[1] P. Maolanon, K. Sukvichai, N. Chayopitak and A. Takahashi, "Indoor Room Identify and Mapping with Virtual based SLAM using Furnitures and Household Objects Relationship based on CNNs," 2019 10th International Conference of Information and Communication Technology for Embedded Systems (IC-ICTES), Bangkok, Thailand, 2019, pp. 1-6, doi: 10.1109/ICTEmSys.2019.8695966.

[2] Open Source Robotics, *About ROS*, ROS.org, Accessed on: August 8 2020 [Online]. Available: https://www.ros.org/about-ros/

[3] C. Pao, *How are Visual SLAM and LiDAR used in Robotic Navigation?,* CEVA, June 11, 2019. Accessed on: August 8 2020. [Online]. Available: https://www.ceva-dsp.com/ourblog/how-are-visual-slam-and-lidar-used-in-robotic-navigation/

[4] B. Gerkey. *Gmapping* , ROS.org, Accessed on; August 8 2020. [Online]. Available: http://wiki.ros.org/gmapping

[5] P. Mihelich, J. Bowman, *cv_bridge*, ROS.org, Accessed on: August 8 2020. [Online]. Available: http://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython

[6] Open Source Robotics, *Integrating ROS with Gazebo*, gazebosim.org, 2014. Accessed on: August 8 2020 [Online]. Available: http://gazebosim.org/tutorials?tut=ros_overview

[7] D. Hershberger, D. Gossow, J. Faust, *rviz*, ROS.org, Accessed on: August 8 2020. [Online]. Available: http://wiki.ros.org/rviz

[8] Ilia G. Polushin, S 2020, Seminar 3: The Unscented Kalman Filter (EKF), lecture notes, ECE 9516B: Topics in Autonomous Robotics-Winter 2020, Western University, delivered 5 Feb 2020.

[9] Ilia G. Polushin, S 2020, Seminar 3: The Unscented Kalman Filter (UKF) Algorithm, lecture notes, ECE 9516B: Topics in Autonomous Robotics-Winter 2020, Western University, delivered 5 Feb 2020.

[10] Robotis E-Manual, *SLAM with ROS*. Robotis.com, 2020. Accessed on: August 8 2020 [Online]. Available: https://emanual.robotis.com/docs/en/platform/turtlebot3/slam/#ros-1-slam

[11] S Nandakumar. *ROS*. Github.com, 2020, Accessed on: August 8 2020 [Online]. Available: https://github.com/Sathyanath42/ROS

[12] Anis Kouba, *ROS for beginners*. Udemy.com. 2020. Accessed on: August 8 2020 [Online]. Available: https://www.udemy.com/course/ros-essentials/

[13] Anis Kouba, *ROS: Navigation and SLAM*, Udemy.com, 2020. Accessed on: August 8 2020 [Online]. Available: https://www.udemy.com/course/ros-navigation/