(Established under Karnataka Act No. 16 of 2013)
100-ft Ring Road, Bengaluru – 560 085, Karnataka,
India

Project Phase II Report

Robotic Process Automation for Industrial Warehouses using Swarm Behavior

Submitted by

SUMANTH V UDUPA(PES1201700525)
PRAMOD KESHAV (PES1201700582)
AJAY VICTOR (PES1201701170)
PRASHANTH B (PES1201701729)
JAN-MAY 2021

Under the Guidance of

Dr. M.J Venkatarangan
Associate Professor
Department of Electrical and Electronics PES University
Bengaluru

FACULTY OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND ELECTRONICS
B.TECH IN EEE



FACULTY OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND ELECTRONICS
BACHELOR OF TECHNOLOGY

CERTIFICATE

This is to certify that the Dissertation entitled

"Robotic Process Automation for Industrial Warehouses using Swarm Behavior"

Is a Bonafide work carried out by

Sumanth V
Udupa(PES1201700525) Pramod
Keshav(PES1201700582) Ajay
Victor(PES1201701170) Prashanth
B(PES1201701729)

In partial fulfillment for the completion of the course work in the Program of Study B.Tech in Electrical and Electronics Engineering under rules and regulations of PES University, Bengaluru during the period January 2021 – May 2021. It is certified that all corrections/suggestions indicated for internal assessment have been incorporated in the report.

(Signaturewithdate&Seal)	*(Signaturewithdate&Seal)*	*(Signaturewithdate&Seal)*
InternalGuide	**xxxxxxxxxxxx**	xxxxxxxxxxxx
xxxxxxxx	Chairperson	DeanofFaculty
Associate Professor	ElectricalandElectronicsEngineering	PESUniversity
ElectricalandElectronicsEngineering		

NameofExaminer:*Signature with date*

1.

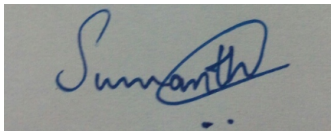
2.

3.

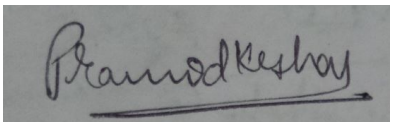
DECLARATION

We Sumanth V Udupa,Pramod Keshav, Ajay Victor and Prashanth B, hereby declare that the project entitled, ***"Robotic Process Automation for Industrial Warehouses Using Swarm Behaviour"***, is an original work done by us under the guidance of Dr. M.J Venkatarangan, Associate Professor, Dept. of EEE, and is being submitted in fulfillment of the requirements for completion of the course work in the Program of Study B.Tech in Electrical and Electronics Engineering.

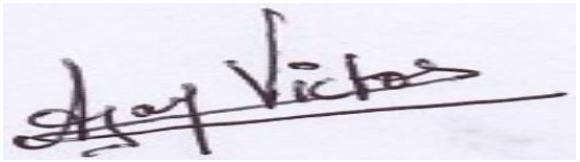
PLACE: BENGALURU
DATE: 12/05/2021



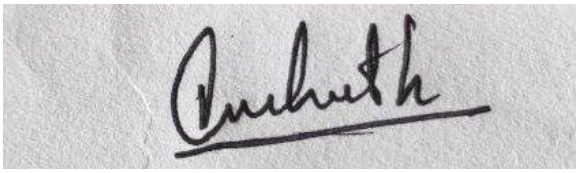
Sumanth Udupa
(PES1201700525)
Electrical and Electronics Engineering



Pramod Keshav
(PES1201700582)
Electrical and Electronics Engineering



Ajay Victor
(PES1201701170)
Electrical and Electronics Engineering



Prashanth B
(PES1201701729)
Electrical and Electronics Engineering

ACKNOWLEDGMENT

We extend our deep sense of gratitude and sincere thanks to our chairman **Dr. M.R.Doreswamy**(Founder, Chancellor –PES University), **Prof. Jawahar Doreswamy,** Pro-Chancellor of PES University and **Dr. J Surya Prasad**, the Vice Chancellor of PES University for giving us an opportunity to be a student of this reputed institution.

We extend our respect to our registrar **Dr. K.S. Sridhar** for his valuable support to conduct this Project under PES institution.

It is our Privilege to thank Chairperson of the Department and Dean of Faculty **Dr. Keshavan.B.K** Department of Electrical and Electronics Engineering for his support and guidance for doing our Project.

We express our sincere gratitude to our guide Dr. **M.J Venkatarangan** for his valuable guidance and suggestion technically and logically for doing our Project work.

We also express our gratitude to all our faculty members, parents and fellow mates who have helped us to carry out this work. Last but not the least, we thank almighty God for his blessings showered on us during this Project period.

ABSTRACT

Swarm robotics is a new approach to the coordination of multi-robot systems which consist of large numbers of relatively simple robots that take their inspiration from social insects such as ants or bees. The most remarkable characteristic of swarm robots is the ability to work cooperatively to achieve a common goal.
Robotic Process Automation abbreviated as RPA is extensively being used nowadays in most of the industries for various processes and applications.

This concept is utilized here for mobile robots which are to be employed in industrial warehouses for efficient task handling. Tasks may comprise of picking up and dropping of goods and packages from the pick-up point to the drop-off point and also sorting of packages/goods.

For our project, we intend to establish a hardware implementation and the working demonstration of a swarm of simple robots that map an unknown environment and also establish autonomous navigation for the robots to move about the warehouse environment and complete the tasks in the warehouse.

In addition, the hardware demo would include a robotic gripper in front of the robots to perform pick-and-place of the packages and goods.

The project had been divided into 2 phases:

Simulation

Simulate the working of a coordinated swarm of robots in a Warehouse environment to showcase the working of:

- Path planning and navigation in a warehouse environment to ensure optimum performance in the warehouse.
- 3D mapping and object localization in any warehouse environment.

Hardware

- Design and development of the multi-robot system.
- Design and development of robotic manipulators.
- Importing the software modules from the simulation section to the hardware and debugging of the same.

In particular, for Phase 2, the hardware implementation of the project has been carried out and will be illustrated in the upcoming sections.

Contents

ABSTRACT 5
1 INTRODUCTION 8
1.1 BACKGROUND 8
1.2 MOTIVATION / PROBLEM STATEMENT 8
1.3 RELATED WORK 8
1.4 OBJECTIVE 10
1.5 ASSUMPTIONS 10
2. EVALUATION OF METHODS 11
2.1 Mapping methods: 11
2.2 Localization methods: 11
2.3 Navigation and Path planning: 12
2.4 Computer Vision: 13
2.5 Control Approach: 13
3. METHODOLODY ADOPTED 14
3.1 Hardware Design and Assembly 14
3.2 Mapping Approach 21
3.3 Localization approach 23
3.4 Computer Vision Implementation 23
3.5 Navigation and Path planning approach 27
3.5.1 Prerequisites for ROS Navigation Stack: 27
3.5.2 Steps to set up ROS Navigation Stack on our Custom Robot: 28
3.5.3 Navigation Stack Implementation 29
3.6 Control Law Implementation 45
3.6.1 Use Case Representation 45
3.6.2 Control Algorithm 45
4. RESULTS 52
4.1 Hardware Assembly and working 52
4.2 Mapping 53
4.3 Localization 53
4.4 Computer Vision 54
4.5 Navigation and Path-planning 55
4.5 CONCLUSIONS AND FUTURE SCOPE 61
5 REFERENCES and LINKS 62

LIST OF FIGURES

Figure 3. 1: Robot Hardware Architecture 14
Figure 3. 2: Mecanum wheel 15
Figure 3. 3: Rotary Encoded DC motor 15
Figure 3. 4: Motor connections 15
Figure 3. 5: ZEDmini Camera 16
Figure 3. 6: L298N Motor Driver 17
Figure 3. 7: Li-ion Cells 18
Figure 3. 8: Li-ion cells in battery holder 18
Figure 3. 9: Omni-directional robot: Possible movements 19
Figure 3. 10: Arduino Mega 19
Figure 3. 11: Raspberry Pi 4 19
Figure 3. 12: NVIDIA Jetson Nano 20
Figure 3. 13:Rosserial comm. between Jetson Nano and Arduino Mega 20
Figure 3. 14: Navigation Stack Setup - Flowchart 28
Figure 3. 15: TF tree for the robot generated through ROS 31
Figure 3. 16: Robot footprint in Rviz 42
Figure 3. 17: Overall Swarm System Flowchart 46
Figure 3. 18: Priority Scheduling algorithm 50

Figure 4. 1: Main robot Hardware (Front view and Top view) 52
Figure 4. 2: Robot connections 52
Figure 4. 3: Map (Occupancy grid) of a different room 53
Figure 4. 4: Map (Occupancy grid) of the environment used 53
Figure 4. 5: Object detection by applying bounding box 54
Figure 4. 6: Object distance shown in terminal 54
Figure 4. 7: Local Costmap visualization in Rviz 55

LIST OF TABLES

Table 1. 1: List of Assumptions 10

Table 3. 1 : Motor Specifications 16
Table 3. 2: Specifications of L298 Motor Driver 17
Table 3. 3: Mapping approaches 23
Table 3. 4: Temporary Obstacle Status 51

Table 4. 1: Odometry State Estimation Table 54
Table 4. 2: Accuracy table without averaging 55
Table 4. 3: Accuracy table with averaging 55

LIST OF ABBREVIATIONS

Cases

AMCL: Adaptive Monte Carlo Localization 23
GPS: Global Positioning System 11
HSV: Hue, Saturation and Value 24
KLD-sampling:Kullback-Leibler Divergence 23
RGB-D: Red Green Blue Depth 21
ROS: Robot Operating System 20
RPA: Robotic Process Automation 8
RRT*: Rapidly-exploring Random Tree Star 12

RRT: Rapidly-exploring Random Tree 12
RTAB: Real-Time Appearance Based mapping 21
SLAM: Simultaneous Localization and Mapping 21
tf: transform trees 27

1 INTRODUCTION

1.1 BACKGROUND

Swarm robots are a collection of similar or dissimilar robots working in close coordination with each other to achieve a common goal. Swarm behavior was first observed in social insects and this attribute has been successfully replicated in a system of robots to improve the performance and efficiency of certain repetitive tasks that would formerly be inefficient using a single robot system.

Although the concept of swarm robots is more widely explored today, it has yet to find any feasible real-world applications. To demonstrate the scalability and the possible applications of this concept in a real-world scenario we have come up with an application titled: Robotic Process Automation in a Warehouse environment using Swarm Behavior.

1.2 MOTIVATION / PROBLEM STATEMENT

In the field of robotics, the application of robots to various kinds of applications is becoming more prominent. Robotics is being employed in almost all the sectors of the industry from supply chain logistics to complex surgeries.

Many industries are solving the problem of warehouse logistics using manipulator/fixed robots as well as autonomous mobile robots. This process of warehouse management is called RPA.
Swarm robotics is a field of robotics which establishes the concept of multiple robots cooperating together to perform single/multiple tasks and make the processes more efficient.

Most of the existing solutions make use of semi-autonomous mobile robots or other such concepts to establish a working model which is relatively less efficient and results in sub-optimal performance.

This project aims to develop a working scalable platform which is able to incorporate a decentralized approach to the swarm behavior that is scalable and portable to other kinds of applications.

1.3 RELATED WORK

The most popular warehouse robotic units are by Tesla, Amazon, 6 River System, Walmart.

- The Amazon warehouse system has robots which only carry and drop the packages. For placing the packages on the robots, a robotic arm is used which has a suction-based end effector. The robot when reaches the destination drops the package with some conveyer type arrangement on it. This system is not intelligent

and is limited to rectangular flat surfaced packages. The complexity in deciding the appropriate approach for coordination among the robots is high because of the huge numbers.

- Tesla uses robots to move heavy parts from one part of the factory to another. These robots use a static map for their path planning and halt when obstacles are encountered. The system is non-intelligent as the robots use line-following techniques along with the map. They have a dedicated battery management system and have an extra feature of charging themselves. Interaction between such robots is minimal in this system.

- 6 River Systems' robot goes by the name Chuck. Chuck is a cobot (collaborative robot), working along with humans to do pick-and-place activities in warehouse fulfillment operations. Chuck directs the pickers to the right items, using platforms with different configurations, shapes and sizes. Chuck approaches the workers in the warehouse with the packages, thus saving time. Shopify acquired 6 River Systems in 2019 to increase its warehouse work, including picking and packing, sorting and inventory replenishment.

These systems are robust but it has not been possible to extract the maximum capacity of the system, due to inefficient distribution algorithms and hardware limitations which are still under development. The main issues faced in the warehouse management are:

1.Managing Warehouse Space/Layout
2.Communication
3.Time management
4.Inventory Accuracy
5.Customer Expectation
6.Redundant activities
7.Product diversification
8.Inaccurate purchase orders
9.Handling product damage

Out of these the main concerns in any average scale, the main aspects in warehouse management that result in loss of time and hence income is Inaccurate Inventory and Incorrect time management.

We propose a robotic automation system that handles these two situations primarily. With the use of robot for the movement of packages and having a proper addressing system the margin of error in inaccurate inventory is greatly reduced. With an algorithmic approach governed by simple laws towards the planning and decision making, the coordination between the robots is improved and the system can perform at a higher capacity.

In support to our current implementation, a thorough study was made on the available techniques/algorithms related to each domain in our project, which has been modified for our use case and efficiency. These approaches along with their

implementation, drawbacks and system limitations are described in the following sections.

1.4 OBJECTIVE

The main objectives of our project are:

- Building a multi-robot system consisting of two robots.
- Development of an autonomous navigation and path planning system for the multi-robot system.
- Design and development of a control algorithm to achieve a coordinated performance and task achievement.
- Integration of a robot gripper for the purpose of pick-and-place of goods/packages by the robots.

1.5 ASSUMPTIONS

For the purpose of demonstration of the deliverables and depicting the robots as a prototype, certain assumptions were made for the complete system implementation.

Assumptions	**Need of said assumption**
The map obtained is of a static environment.	The packages are not present in the initial map of the environment and are only detected during tasks.
Packages in the environment are assumed to be uniformly distributed.	This is to make sure that packages are not placed randomly in the environment with concentration in only a specific region.
Warehouse is assumed to be a rectangular room.	There will be an even distribution of the workspace environment between the members of the swarm.
We have a set number of pickup and drop off buffers at any point of time in the environment.	This ensures that every package in the environment is accounted for and is not lost or misplaced.

Table 1. 1: List of Assumptions

2. EVALUATION OF METHODS

This section gives a general description of the various technical domains that are implemented in the hardware.

2.1 Mapping methods:

Map building by exploration of an unknown environment is a crucial step for the implementation of autonomous navigation of a mobile robot. This is one of the initial steps that has to be performed by a robot for navigating through an environment and also for efficient path planning to perform various tasks and operations.

Prerequisites and Conditions for a map:

- The precision of the map must match the precision with which the robot needs to achieve its goals.
- The type of features represented must match the precision of the features obtained by the sensors onboard the robot.
- The complexity of the map representation has a direct relationship with the computational complexity of localization, navigation and path planning.

There are two types of maps: discrete and continuous. The maps generated by navigation generally provide a continuous geometric description. This usually needs to be converted to appropriate discrete type maps for the implementation of path planning algorithms.

2.2 Localization methods:

When the robot is moving for long distances, the robot must determine its position relative to an external reference which is often called a ***landmark***. This process is termed as ***localization***. These landmarks are obtained from the static map obtained by the robot during the mapping process. So, when the landmark is in view, the robots localize frequently and accurately, using action and perception (from sensors) update to track their position. Thus, the robot is effectively navigating from one landmark zone to the next, with the utilization of localization.

Basically, there are two methods of localization present to localize any autonomous mobile system. One is **global/absolute positioning** and the other is **local/relative localization**. Our objective was to use relative positioning to determine the robot's position using the sensors present onboard the robot and rely less on the information from its surroundings.

- Global positioning:

It helps to determine the position of the robot w.r.t coordinates that already have a reference i.e., a global coordinate. GPS is most commonly used for this and is accurate.

But for indoor activities, it is not suitable for our project and can be used

only outdoors because GPS receivers need an unobstructed view of the sky. We are subjecting our robots to work only in indoor environments presently. The robot hardware can be modified for different environments and surfaces.

- Local positioning:

  It determines the position of the robot w.r.t its movement from its initial position. It does not take into account any data from its environment. It evaluates the position using various on-board sensors like encoders, gyroscopes, accelerometers, etc. In our case we will be using wheel encoders present on the robot hardware to obtain the position and orientation respectively.

2.3 Navigation and Path planning:

Navigation is an essential step that is directly linked to the robust mobility of the robot. When only partial knowledge of the environment is provided along with a goal position or series of positions, navigation approaches/algorithms provide the ability of the robot to act based on its knowledge of the surroundings and the sensor values to efficiently reach the goal position.

Robot path planning involves the problem of moving from one place to another, simultaneously performing tasks prescribed the user. The algorithm developed for this should be capable of computing a collision-free path between a start point and a goal point. Collision-free indicates the prevention of collision between other dynamic obstacles (robots, people, cars, etc...) and also static obstacles which are fixed in the workplace or the environment in which the robot is present.

Path planning can be either global or local.

Local path planning is the process of planning a path when the robot is moving while taking data from the sensors onboard the robot. During local path planning, the robot can account for sudden changes in the environment, which may occur due to other robots or people moving around which is captured from the robot's sensors.
Global path planning is performed only when the environment is static and it is known to the robot i.e., the robot previously has the map of the environment in its memory. For this type of path planning, the algorithm produces a complete path from the start point to the goal/target position. This is done through a set of waypoints provided to the robot, even before the robot is in motion.

There are various path planning algorithms and this is currently a research area with more efficient algorithms being introduced.
But the most prominent and used path-planning algorithms are:

- Dijkstra algorithm
- A* path planning algorithm
- RRT and RRT* algorithms

For the implementation of a path planning algorithm onto a mobile robot, the

combination of both local and global path planning is necessary and each of those approaches uses a combination of the above algorithms.

2.4 Computer Vision:

This domain is mainly responsible for the vision aspect of any autonomous system and to provide perception of the robot's environment.
For an autonomous mobile robot, the vision sensors i.e., the camera is mainly responsible for providing a constant feed to the robot about its environment and aid it in navigating and completing tasks around the environment.

Computer vision allows a computer to process the image provided to it and gain a certain level of understanding and relevant information for it. With the large number of open-source libraries and resources computer vision has found great use in a wide range of applications and fields as per the user's requirement.

- OpenCV:

OpenCV is an open-source library available to develop computer vision-based applications as per a user's requirements. OpenCV allows real-time implementations of image processing, computer vision and machine learning concepts to identify various features of an image and use the information obtained to provide the system a certain level of understanding and information as deemed relevant to perform its final task.
A large number of applications such as face detection and recognition, security and surveillance, object recognition etc. make use of OpenCV.

- Object detection and localization:

Object detection and object localization are two major aspects in computer vision that allows us to identity objects and its boundaries and locate an object in an image respectively.
Object detection is the process of finding an object of interest in real time in a real-world scenario. There are multiple methods of implementing object detection out of which the simplest and most commonly used methods are detection based on *shape* and *color*. These methods help us keep a track of every instance of an object of interest in a given image frame.
Object localization is the process of locating a specific instance of an object in an image. Unlike object detection which keeps a track of all the instances of an object's occurrence, object localization aims to locate the main and most prominent occurrence in the current image frame of reference.

2.5 Control Approach:

Design of a multi-robot system is usually based on the following two structures:

- Centralized structure:

  This is a system has a robotic agent (a master/leader) that is in charge of organizing the work of the other robots, which are referred to as slaves/followers. The master is involved in the decisional process for the

whole team, while the other members act according to the direction of the leader.

- Decentralized structure:

This is also known as Distributed Control.
This is a system composed of robotic agents which are completely autonomous in the decisional process with respect to each other, in this class of systems a leader/follower relationship does not exist.
A centralized control structure is not feasible, because there are a large number of individual robots with limited sensing capabilities. Distributed control is required for flexibility and reliability. This is a way of distributing control to certain regions of swarm so that any effective control that should be taken is limited to the affected region/neighborhood.

However, both distributed and centralized control approaches have contributed individually to the study of swarm robotics and have generated interesting experimental results.
Combination of both the different control structures will yield good results in implementation.

3. METHODOLODY ADOPTED

This section describes the various approaches we have considered for our implementation and the methods we have finally implemented for the working of the multi-robot coordinated system.

3.1 Hardware Design and Assembly

The hardware is comprised of two four-wheeled robots. The hardware architecture for the connections onboard the robot is depicted through the figure given below:

ROBOT HARDWARE ARCHITECTURE

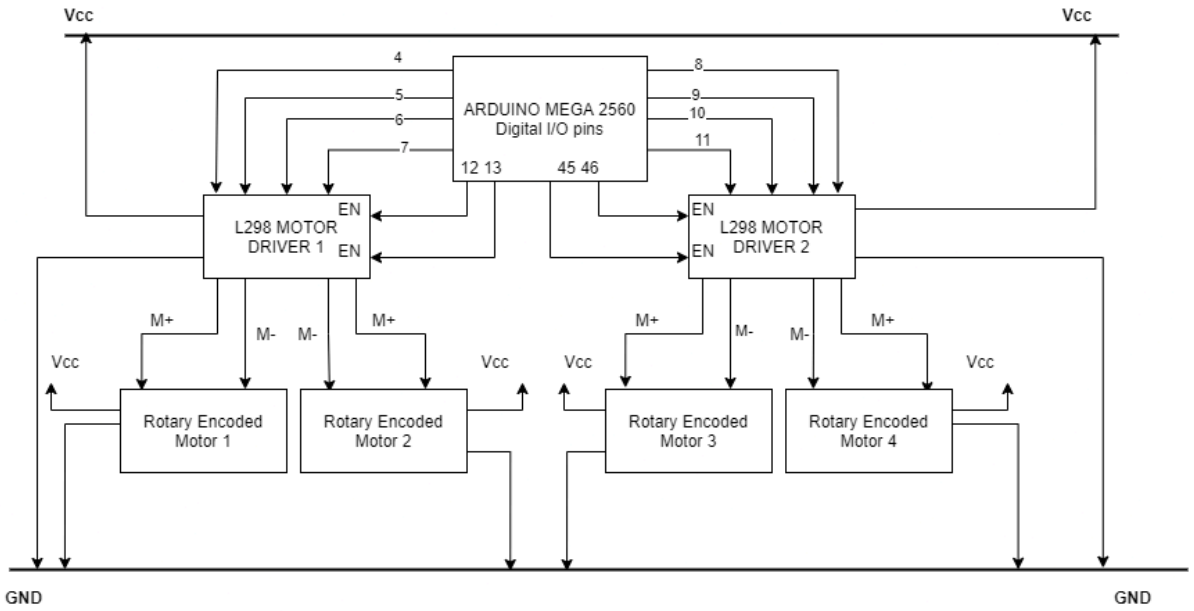


Figure 3. 1: Robot Hardware Architecture

The details of the robot hardware developed are provided below.

- **Mecanum Wheels**

Model name: 60mm Aluminum Mecanum wheel set



Figure 3. 2: Mecanum wheel

These are also known as Swedish wheels and are omnidirectional i.e., the wheel can move in any direction. The wheel consists of many rollers which typically have a 45^0 axis of rotation with respect to the plane of the wheel. These wheels were used because the robot would be stable and be capable of moving in a combination of 8 different directions.

- **Sensors used**

i. Quadrature encoded motors:

Model name: RMCS-2295 Quad encoder motor.



Figure 3. 3: Rotary Encoded DC motor



Figure 3. 4: Motor connections

For the purpose of obtaining odometry information which is required for localization(dead-reckoning), quadrature-encoded motors were used.
So, for two robots, a total of 8 encoded motors were required. This provided a speed of 150 rpm and required a voltage of 12V for operation.
The quadrature encoders helped in obtaining the direction of the motor and also the distance travelled by each motor. The encoders provided a total of 280 pulses per channel. There are six connections present on each motor.
They are:

- Motor+
- Motor-
- Vcc(supply) and Ground
- Channel A
- Channel B

The channels provide two pulses each, where there is a phase shift of 90^0 between Channel A and B. Based on the leading pulse, the direction of rotation of the motor can be determined. Depending on which channel gets the encoder pulses, it generates an interrupt in the Arduino Mega microcontroller and we can detect the direction of rotation based on which, other computations are performed.

Motor specifications are as follows:

Operating Voltage	12V
Load current(max)	1A
No-load current	140mA
Stall torque	8 kg-cm
Rated torque	3.2 kg-cm

Base motor speed	7000 rpm
No-load speed	150 rpm
Encoder pulses per channel	280

Table 3. 1 : Motor Specifications

ii. ZED Mini camera:



Figure 3. 5: ZEDmini Camera

The ZED mini is a stereo camera that provides conventional images as well as an accurate depth measure of its immediate surrounds. This camera was designed for applications such as real time environment mapping, security and surveillance, Ariel's drone autonomous navigation and mapping …etc.
The ZED mini is a very versatile and portable camera with a very high-resolution video output and also includes motion sensors such as Gyroscope and Accelerometer along with Depth sensing which allows for both Static environment mapping and visual-inertial localization due to which it finds great applications in SLAM based robotics projects.
The ZED also has software support in terms of the ZED SDK which contains all the drivers as well as libraries required to use the ZED along with the ZED API that allows manipulation of the camera parameters such as frame rate, resolution …etc. as per the user's requirements.
In our application the ZED has been used to effectively map the robots' surroundings as well as for object detection and localization where the main advantage of the ZED camera is its ability to accurately determine and return the distance of an object from its current position.

iii. Motor driver:

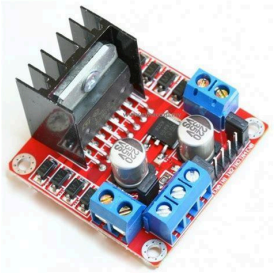


Figure 3. 6: L298N Motor Driver

The microcontrollers used could not provide the necessary current to drive the motors. So, for this purpose, motor drivers were used. In specific, the L298 Dual H-Bridge motor drivers which are bi-directional were used for this

purpose. It allows easy and independent control of two motors of up to 2A each in both directions. It comes equipped with power LED indicators, on-board +5V regulators and also pins for PWM control. With the enable pins, the speed of the motors could be varied by changing the duty cycle of the input signal to the driver.

So, a total of four (2+2) motor drivers were used to drive the robot motors. The supply for the drivers was provided by the Li-ion cells.

Specifications:

Input voltage	3.2V ~ 40V dc
Peak current	2A
Operating current range	0 ~ 36 mA
Control signal input voltage range	Low: -0.3V ≤ Vin ≤ 1.5V High: 2.3V ≤ Vin ≤ Vss
Enable signal input voltage range	Low: -0.3V ≤ Vin ≤ 1.5V (invalid control signal) High: 2.3V ≤ Vin ≤ Vss (control signal active)

Table 3. 2: Specifications of L298 Motor Driver

iv. Battery:



Figure 3. 7: Li-ion Cells

Power supply for the entire robot was provided using two methods. The microcontrollers onboard were powered from power banks(4800mAh).
A combination of 3 Li-ion cells (18650) were connected in series by a battery-holder and they provide a total of 12V to run the motors for the robot. Each Li-ion cell is 18mm around 65mm long and has a capacity of 3.7V and 2000mAh and are rechargeable.



Figure 3. 8: Li-ion cells in battery holder

v. Chassis:

To realize the omnidirectional motion of the robot, a 4-wheeled chassis was used. The chassis was metallic and 29cms long and 17.5cms wide. Four metallic clamps were used to hold the motors which were fixed to the bottom of the robot. Arduino Mega and the batteries were also attached to the bottom of the robot using screws and Velcro straps respectively.

The cameras for the two robots that were used for mapping/sensing/object detection were placed on the top of the chassis. To obtain more accurate readings from the camera, a frame was embedded on the robot using small metallic stands an acrylic frame for placing the camera.

To switch the Li-ion cells ON/OFF, a switch(5A/220V) was used.

This figure depicts the possible directions of movement of the robots.

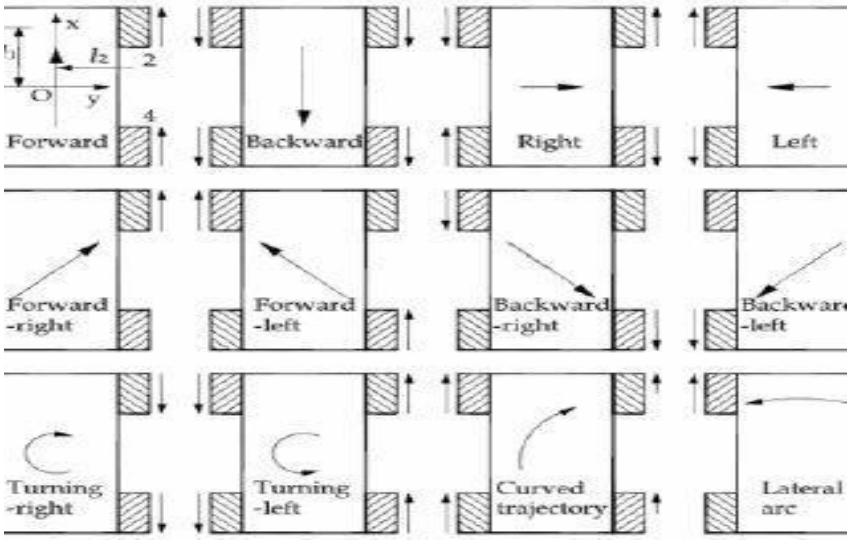


Figure 3. 9: Omni-directional robot: Possible movements

- **Microcontrollers**

i. Arduino Mega:

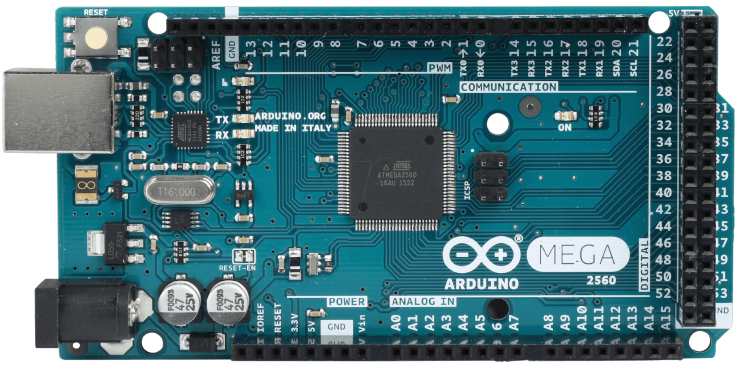


Figure 3. 10: Arduino Mega

The odometry data is required for robot navigation, which is obtained using the motors with hall sensors. For this interface we need a microcontroller which is capable of handling such data from 4 different motors. Arduino Uno has only 2 external interrupt pins, but in our case, we needed 4. So, we are using the Arduino Mega which has 6 interrupts available.

ii. Raspberry Pi 4:



Figure 3. 11: Raspberry Pi 4

The Pi4 has a powerful processor and it meets our criteria for managing computation of data from the localization, path planning and object detection modules. It receives the encoder data from the Arduino and acts as a master controlling the Arduino thereby controlling the motors.

iii. NVIDIA Jetson Nano:

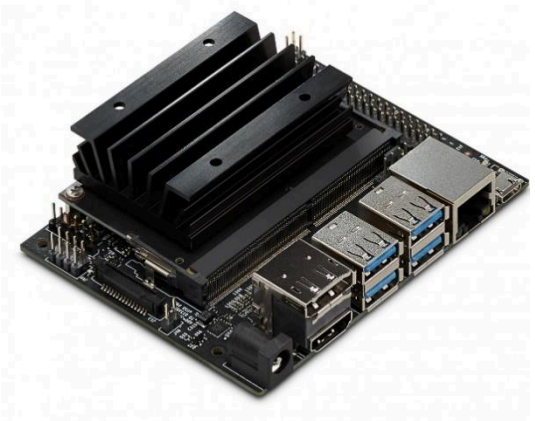


Figure 3. 12: NVIDIA Jetson Nano

The Pi4 has enough processing capacity for the tasks mentioned above but it

lacks the necessary GPU which is required for area mapping and object detection and localization. NVIDIA Jetson Nano fulfils this requirement and an additional advantage with this Microcontroller was the CUDA libraries that were readily available for the ZED mini camera. This is used as the master and takes in encoder data from Arduino mega and handles other tasks such as localization, path planning etc... This microprocessor was interfaced with the primary robot which was used for mapping out the environment.

- Communication using ROS:

Communication between the Raspberry Pi 4 and Arduino Mega is done via Serial BUS as a physical medium with the use of *rosserial_arduino* library. *Rosserial* is a protocol for wrapping standard ROS serialized messages and multiplexing multiple topics and services over a character device such as a serial port or network socket.
So, in overview we have successfully set up ROS in Raspberry Pi 4 as well as in the Jetson Nano, and established the *rosserial* library for the communication between the Pi 4 and Mega and also between Jetson Nano and Mega.

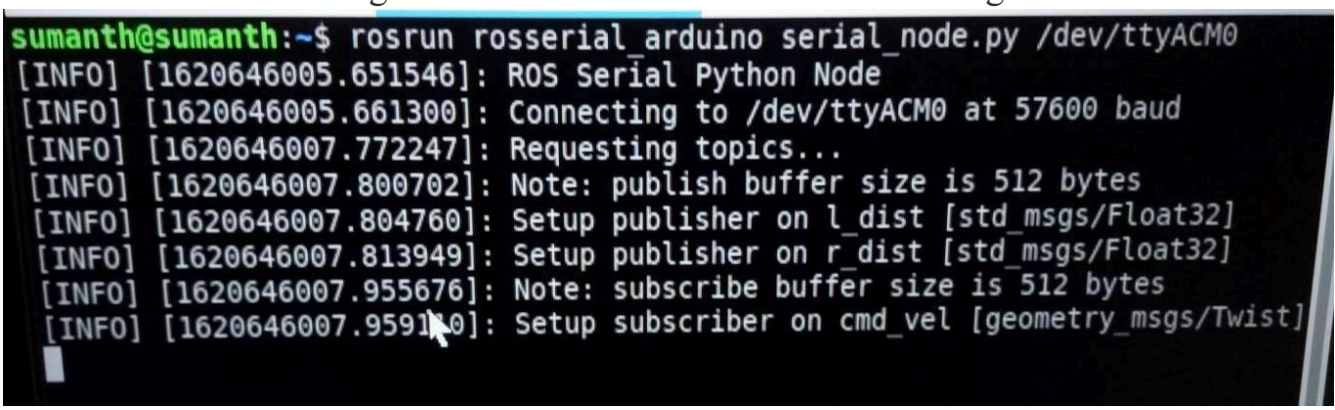


Figure 3. 13:Rosserial comm. between Jetson Nano and Arduino Mega

Throughout the report, the Jetson Nano and the Raspberry Pi 4 will be collectively called as the main processor because the same software and code is portable to both the robots.

3.2 Mapping Approach

Since ROS was present on both the robots, it provided many approaches that could be used for mapping which were compatible with ROS.
Some of the approaches that have packages in ROS are: SLAM using 2-D and 3-D mapping, Real-Time appearance-based mapping (RTAB map) and also Octomap. Since we possessed a ZED Mini camera which is a depth camera, we decided to pursue the concept of 3-D mapping through which we could obtain a real-time appearance-based map that could be better utilized by the robot for localization.
Some of the approaches are described below:

- SLAM: Simultaneous Localization and Mapping is a famous approach that is

under constant research and development in the robotic society.
It deals with the problem of leaving the robot in an unknown environment and for the robot to build a map incrementally by moving around the environment and determining its location or position within this map.

The SLAM approaches with 3–D mapping that we pursued were ORB-SLAM2, RTABmap and Octomap.

- ORB-SLAM2: It is a real-time SLAM library for Monocular, Stereo and RGB-D cameras that compute the camera trajectory and a sparse 3D reconstruction of the environment. It is highly efficient and is able to detect loops and re-localize the camera in real time. It also had options to run the package in only Mapping mode; Localization mode or in SLAM mode. We were able to obtain the 3D map but the main disadvantage for us was that it required high computation power due to which there was high latency in obtaining the map and we could not obtain the map in real-time. Hence, this method was dropped.

- RTAB map: The next approach we considered was RTABmap which could obtain a real-time appearance-based map of the environment similar to a live camera feed. This approach is based on an incremental appearance-based loop closure detection. The loop closure detection helps in detecting repetitive images obtained from the camera when the robot is moving and omits such repetitive feed during the rendering of the 3D map. This was the main premise behind choosing this approach for mapping. But we could not render a map with this approach, since it required a minimum of 4gb RAM and i7(or equivalent) processor for the mapping approach to run seamlessly.

Thus, this approach was also discarded.

Since both these approaches required a lot of computation power, we decided that we would implement mapping and localization separately and establish a connection between the two by providing the map for the localization algorithm.

The next most viable approach was Octomap which is an efficient Probabilistic 3D mapping framework based on Octrees. It obtains a Full 3D model of an arbitrary environment without pre-assumptions and also covers occupied areas as well as free space.
We were able to obtain a good real-time appearance-based map of the environment at a standalone perspective, but when we integrated the mapping approach with the remaining modules, the results obtained were not very satisfactory and could not be expected to provide meaningful outputs in the long run.
So, we had to discard this approach for mapping and search for an approach which used reasonable computation and provide good results.
Thus, after our literature survey we decided to implement Google Cartographer.

Cartographer is a system that provides real time SLAM in 2D and 3D for different sensors and is supported by ROS systems. It uses the information about the robot's surroundings from the sensors onboard and builds a map of the environment and helps the robot to localize itself in the map.
But we are using the Cartographer approach solely to generate the map of any environment where the robot is subjected to and using a different localization approach described in Section 3.3
During the implementation of the Navigation Stack, we learnt that the Stack requires a 2-D occupancy grid or cost map as an input for the path planning algorithms to provide an optimal path.
Because of this, we decided to use a package called **depthimage_to_laserscan** present in ROS that converts a depth image to a set of laser scans in real-time so that the Navigation Stack can use the map directly.

The table below provides a summary of the different approaches considered and their merits and demerits:

Mapping approach	ORB-SLAM2	RTAB map	Octomap	Cartographer
Type of map	Key point feature-based map	Appearance based map	Based on 3D occupancy grid using octrees	2D map: Occupancy grid
Requirements	i7(or above). Depth camera C++ compiler Pangolin library OpenCV library	i7(or above) Depth camera	C++ compiler	ROS Kinetic and above. 2GB RAM and above for good results.

	ROS			
Suitability	Computationally intensive for our use case. High latency in real-time	Computationally intensive and low frame rate when used in Jetson Nano or Raspberry Pi 4	Puts load on computation and not easily integrable	Works well on our current system with minimal latency. Most compatible than all other approaches considered

Table 3. 3: Mapping approaches

3.3 Localization approach

Since mapping was done separately, localization was implemented using a typical approach which was compatible with ROS.
AMCL known as Adaptive Monte Carlo Localization is a well-known established approach used for localization of robotic platforms.
It is a 2-D probabilistic approach used for moving a robot. It uses a particle filter, in particular, the adaptive KLD-sampling approach for tracking the position of the robot with respect to the static map provided by the robot.

 Monocular Odometry
Monocular odometry is an approach used to calculate the position of a robot over time in reference to its initial position based on visual data provided via a monocular camera. This system is typically used in autonomous robots and estimates the change in position based on a scale factor that estimates the change in size of objects in current image frame with reference to the previous frame or based off of the data from other sensors such as an IMU or wheel encoders.

We had initially considered implementing monocular odometry to improve the odometry reading obtained from the wheel encoders so as to account for the inaccuracies encountered due to drift in terms of the wheel encoders. But since monocular odometry would require estimated x, y coordinates from the wheel encoders in real time with reference to each image frame, this implementation would be redundant and the errors would persist and be carried over.
Estimation of odometry using only a scale factor estimated based of object size in image frames would also lead to a large amount of drift in estimated values based on change in camera orientation with respect to the reference object and give rise to errors.

Due to the above-mentioned limitations and redundancies faced we decided to not implement visual odometry and instead use the wheel encoder based odometry approach.

3.4 Computer Vision Implementation

The main application of Computer Vision in our project is for Object detection and localization in the robot's immediate surroundings.

Here we make use of the computer vision library OpenCV along with the NumPy library which is used to work with arrays on python. Here OpenCV with NumPy is used for object detection based on color and shape as well as object localization in the image frame to obtain the x, y coordinates of the object in question with respect to the image frame.

The distance of the object from the camera is obtained with the help of the point cloud generated by the ZED mini which represents a set of x, y, z data points in space and can be used to estimate the approximate distance between the camera and the object.

- Color and Shape detection

OpenCV has built-in libraries to help in both shape and color-based object detection.

Here, for our implementation of color detection we have specified the range of **HSV** values in 2 **NumPy** arrays to create a mask of a color of our choice, this mask is then applied to our image frame using a bitwise AND operation to obtain our image of interest as per the specified color.

```
frame = image_zed.get_data()
depth_image_ocv = depth_image_zed.get_data()
frame = cv2.cvtColor(frame,cv2.COLOR_BGRA2BGR)
#cv2.imshow("Image", frame)
#cv2.imshow("Depth", depth_image_ocv)
#_, frame = capture.read()
#frame = cv2.imread("/home/ubuntu/objects_detect.jpeg")
# Converts images from BGR to HSV
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
lower = np.array([0,50,50]) #red
upper = np.array([10,255,255])
# Here we are defining range of bluecolor in HSV
# This creates a mask of blue coloured
# objects found in the frame.
mask = cv2.inRange(hsv, lower, upper)
# The bitwise and of the frame and mask is done so
# that only the blue coloured objects are highlighted
# and stored in res
res = cv2.bitwise_and(frame,frame, mask= mask)
cv2.imshow('frame',frame)
cv2.imshow('mask',mask)
cv2.imshow('res',res)
```

The above code shows the use of **cv2.bitwise_and** on the image frame obtained by the **ZEDmini** camera to obtain the resultant frame ***res*** which only displays objects of a specific color as determined by the NumPy array specified by the variables **lower** and **upper** that determines the HSV thresholds of the color.

Since color detection alone may not give accurate detection of objects in the environment as there may be various objects in the same color, we have implemented shape detection to ensure a more accurate detection of the objects of interest. We have assumed our objects to be rectangular in shape and have used ***Canny edge detection*** to determine the contours of the objects within a specified range of edge intensity gradient.

```
cv2.namedWindow("Parameters")
cv2.resizeWindow("Parameters", 640,240)
cv2.createTrackbar("Threshold1", "Parameters", 20, 400, empty)
cv2.createTrackbar("Threshold2", "Parameters", 20, 400, empty)
```

The code snippet above is used to create 2 track bars to vary the thresholds of edge intensity gradient as per our object to be detected.

```
threshold1 = cv2.getTrackbarPos("Threshold1", "Parameters")
threshold2 = cv2.getTrackbarPos("Threshold2", "Parameters")
imgCanny = cv2.Canny(imgGray, threshold1, threshold2)
kernel = numpy.ones((5,5))
imgDil = cv2.dilate(imgCanny, kernel, iterations=1)
imgContour = res.copy()
getContours(imgDil, imgContour)
#cv2.namedWindow("frame")
#cv2.setMouseCallback("frame",Click)
cv2.imshow("found", imgContour)
```

Once Canny edge detection has been implemented on a grayscale image of our frame, we apply a 5x5 image kernel to perform dilation to increase object area and also accentuate the edges detected by Canny and the image after dilation and a copy of the original image are passed as inputs to the **getContours** function.

```
def getContours(img, imgContour):
    contours, hierarchy = cv2.findContours(img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

    for contour in contours:
        area = cv2.contourArea(contour)
        if area > 3000:
            #cv2.drawContours(imgContour, contour, -1, (255,0,255), 7)
            points = cv2.approxPolyDP(contour, 0.02*cv2.arcLength(contour, True), True)
            numSides = len(points)
            #print(numSides)
            #if numSides == 4:
            x, y, w, h = cv2.boundingRect(points)
            cv2.rectangle(imgContour, (x, y), (x+w, y+h), (0,255,0), 5)
            #print("x:",x,"y:",y)
            #Distance of object
```

The **getContours** function uses the function **cv2.findContours** dilated image to obtain all the contours of all the objects as present in the image, we may also specify the area under consideration using the **cv2.contourArea** function.

The **approxPolyDP** function is then used to obtain the approximation of all the points in the image that may make up a polygon and these points are then passed on to the **cv2.rectangle** function to draw a rectangle boundary on the objects of interest.

We now approximate the points that make up a polygon by using the **cv2.approxPolyDP** and once estimated these points may be used to draw a bounding rectangle on the original image as shown above using the **cv2.rectangle** function.

The estimation of object distance from the camera has been made possible due to the use of the Point Cloud data provided by the ZED mini camera.

```
x, y, w, h = cv2.boundingRect(points)
cv2.rectangle(imgContour, (x, y), (x+w, y+h), (0,255,0), 5)
```

```
#Average distance
i=int(x+(w/2))
j=int(y+(h/2))
x_mid=i
y_mid=j
dsts = list()
cnt=8
t=-1
while(cnt):
    err, point_cloud_value = point_cloud.get_value(i,j)
    dt =  math.sqrt( point_cloud_value[0] * point_cloud_value[0] + point_cloud_value[1] * point_cloud_value[1] + point_cloud_value[2] * point_cloud_value[2])
    dsts.append(dt)
    if(cnt%2==0):
        i=int(i+(t*w/70))
    else:
        j=int(j+(t*h/70))
    cnt=cnt-1
    t=t*-1

distance= sum(dsts)/len(dsts)
point_cloud_np = point_cloud.get_data()
point_cloud_np.dot(tr_np)
```

The x, y coordinated in the image frame can be estimated with reference to the bounding rectangle drawn on the image after shape detection and these x, y coordinates are passed as inputs to the **point_cloud.get_value()** function that allows estimation of the x, y, z points in that point in space. The function **cv2.boundingRect(points)** returns the starting coordinate (x, y) of the rectangle and its respective width and height **(w, h)**. For better accuracy the distance **z** is calculated by averaging a total of 9 values from various neighbors around the center of the rectangle. The z value is computed for the center point of the rectangle and also for 8 distinct neighbors. The distance between the center point and the neighbors can be varied as suitable for the test case. Empirically a value of 70 or above is recommended. The value selected is used to divide the dimension into that many parts and take the average distance of these values. This way we can avoid the error which can occur due to change in horizontal alignment w.r.t the camera view.

```
if not np.isnan(distance) and not np.isinf(distance):
    print("Distance to Camera at ({}, {}) (image center): {:1.3} m".format(x, y, distance), end="\r")
    point_publisher(x_mid,y_mid,distance)
else:
    print("Can't estimate distance at this position.")
    print("Your camera is probably too close to the scene, please move it backwards.\n")
sys.stdout.flush()
```

3.5 Navigation and Path planning approach

Initially, our approach was to develop an independent module for path planning in Python programming language which was developed during the simulation phase and was based on A* path-planning algorithm.

The input for the code would be a map of the environment in pgm format and the output would be a set of waypoints written to a csv file that the robots should follow to reach the goal position.

But this approach could not be used here, since we had to take into account a

lot of factors such as slip of the wheels, payload on the robot, etc... that were not considered during the simulation phase.

So, an alternative was to use the ROS Navigation Stack which is described in detail below:
The Navigation Stack takes in odometry and sensor streams as inputs and provides the velocity commands as output that are fed to the motors present onboard the mobile robots through Arduino Mega acting as a low-level controller.

3.5.1 Prerequisites for ROS Navigation Stack:

- The robot must be running ROS.
- The robot should have a transform tree(tf) setup.
- Sensor data should be published using the correct ROS Message types.
- It needs to be configured for the shape of the robot.

The below figure taken from the ROS wiki page illustrates the configuration of the Navigation Stack:

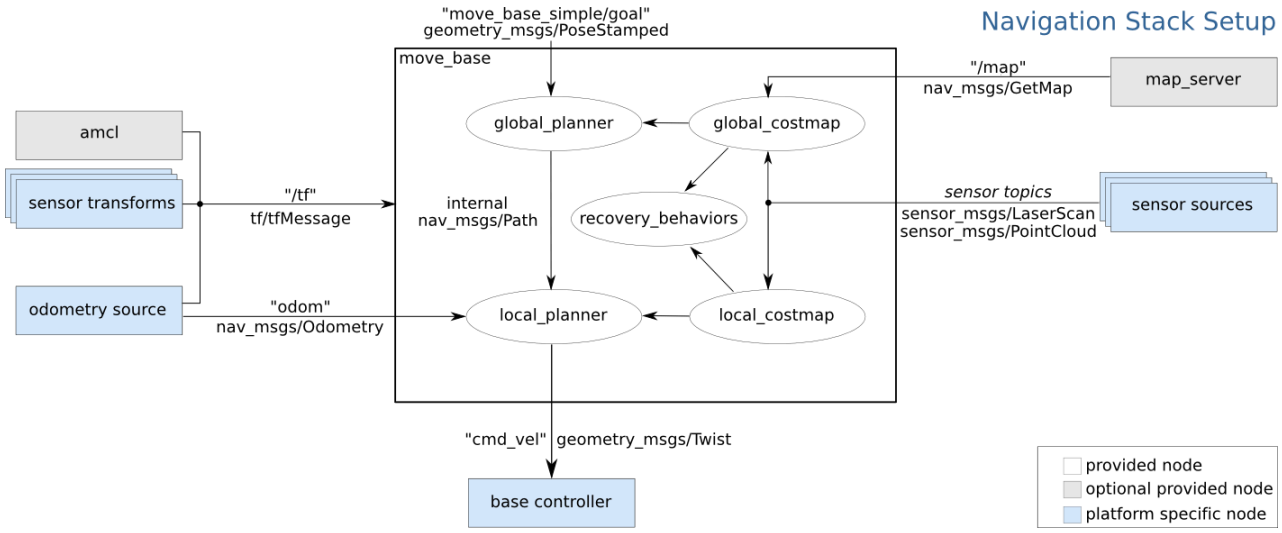


Figure 3. 14: Navigation Stack Setup - Flowchart

3.5.2 Steps to set up ROS Navigation Stack on our Custom Robot:

The below five steps are the mandatory pre-requisites for the ROS Navigation Stack Setup. Each step was developed independently and tested and integrated finally to launch the entire ROS Navigation Stack.

1. Set up transform trees(tf) on the robot
   - tf will take care of conversion of frames i.e., conversion of coordinates from the camera_frame to the base_link of the robot and also to obtain coordinates of the objects in the world_frame and pass it to the base_link of the robot.
2. Publishing sensor streams over ROS
   - Using this concept, can directly publish the 3-D point clouds obtained from the ZED Mini to the Navigation stack through ROS.

- Similarly, can publish data from sensors (monocular camera) onboard the 2nd robot.

3. Publishing odometry information over ROS
   - Similar to publishing sensor streams, programmed through ROS to send odometry information from the wheel encoders to the navigation stack.
   - Using this .cpp file, the navigation stack can directly send velocity commands to the "cmd_vel" topic. A node will be subscribed to this topic which in turn takes the values **(vx, vy, vtheta) <==> (cmd_vel.linear.x, cmd_vel.linear.y, cmd_vel.angular.z)** velocities and converts them into motor commands to send to the Arduino Mega which will control the motor speeds through PWM for moving the robot.
   - This part of the Navigation Stack is platform dependent. So, our approach here was, we developed our own code in the odometry file so that the we could send the velocity commands to the robot wheels through the same file.
4. Mapping the environment
   - Built a map using Google cartographer to map the environment and used **depthimage_to_laserscan** package to obtain the 2D **costmap** required for path planning. Later, loaded this generated map as an input to the Navigation Stack.
5. Navigation Stack setup
   - Sending multiple goal points to the Navigation Stack through waypoints.
   - Using this, can complete path planning which is taken care of internally by the navigation stack which has inbuilt **costmaps*** (global, local) and also **local_planners** (DWA) and **global_planners**(A*).

   *Costmap: It represents a 2D map for navigation in terms of an occupancy grid to calculate the obstacle areas, free space and unknown space.

   It is expressed as a value between 0 ~ 255, where:

   0: Free space allowed for the robot to navigate

   1~127: Areas of low collision

   128~252: Areas of high collision

   253~255: Occupied areas where the robot cannot move
   - Navigation Stack Tuning: This is the last step of the process which can be performed to tune the parameters of the nav stack to get better results in path planning and mapping.

Using these steps, we could setup the Navigation Stack on our custom robot.

3.5.3 Navigation Stack Implementation

There are two microcontrollers/processors for each robot. The main robot has a Jetson Nano that is running ROS Melodic and an Arduino Mega which acts as the controller for the **Mecanum** wheels.

Programming on ROS Melodic was done mainly in C++ and Python, whereas Embedded C++ was used for Arduino Mega.
The upcoming sections describe the implementation codes developed for autonomous navigation of the robot.

1. Setting up transformation tree(tf) for the robot:

Usually, for a robot, there will be many coordinate frames corresponding to the various frames onboard the mobile platform. ROS uses the concept of **broadcaster** and **listener** to transmit data between the different frames of the robot.
The notations used for the coordinate frames for our robot were:

- **map:** This is the world frame. The environment where the robot is functioning is depicted as the world frame and all other transformations will be with respect to this frame.
- **odom:** This frame indicates the pose of the robot in relation to the map frame i.e., as the robot moves, it is linked to the **odom** frame which in turn is a child frame of the map frame.
- **base_link**: This is the coordinate frame which is rigidly attached to the base of the robot. We have chosen the midpoint of the robot chassis as the base frame and all the frames atop the robot are specified with respect to this frame.

```
while(n.ok()){
  broadcaster.sendTransform(
    tf::StampedTransform(
      tf::Transform(tf::Quaternion(0, 0, 0, 1), tf::Vector3(0.035, 0.0, 0.052)),
      ros::Time::now(),"base_link", "base_camera"));
  r.sleep();
 }
}
```

This code snippet shows the transformation between the parent frame "**base_link**" of the robot to the child frame "**base_camera**" atop the robot. The **tf::Transform** object that is present in the tf package has 5 arguments. The **tf::Quaternion** specifies the rotation between the parent frame and the child frame. For our case, since there is no rotation, the Roll, Pitch and Yaw are made 0 for the Quaternion. The **tf::Vector3** is used to specify the offset between the two frames. The units are in meters, so the x-offset is provided as 0.035m(3.5cm) from the base frame and the z-offset i.e., the upward distance of the camera from the robot is given as 0.052m(5.2cm). **ros::Time::now()** updates the time stamp dynamically between the two frames when the Navigation Stack is running.
The tf tree for our robot is shown below:

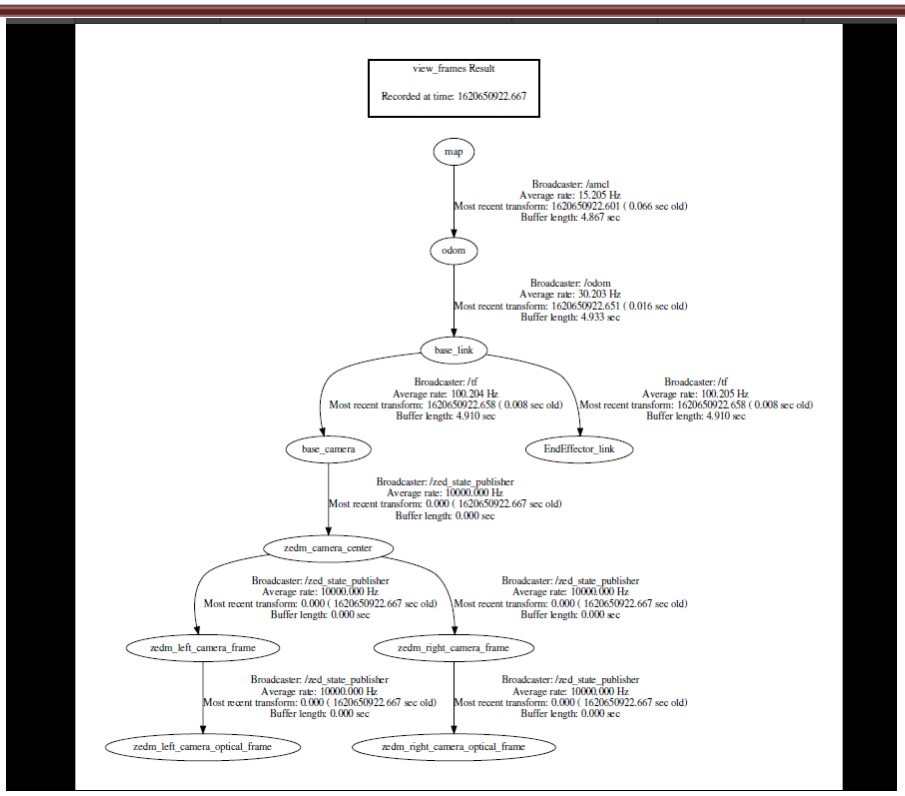


Figure 3. 15: TF tree for the robot generated through ROS

2. Publishing Sensor streams over ROS:

Navigation Stack requires sensor information for navigating around an environment and also for localizing itself in the environment. We are using a ZED mini camera as the sensor and the detailed implementation procedure is provided in Section 3.4

The Navigation Stack by default takes inputs of the type: **sensor_msgs/LaserScan** and **sensor_msgs/PointCloud.** Since the **ZEDmini** was not publishing this topic, we used a package "**depthimage_to_laserscan**" to convert it to the **LaserScan** topic.

The **depthimage_to_laserscan** package takes a depth image provided by the **ZEDmini** camera and generates a 2D laser scan based on the provided parameters. It subscribes to the **rostopics sensor_msgs/Image** and **sensor_msgs/CameraInfo** and publishes the scan topic: **sensor_msgs/LaserScan**.

3. Publishing odometry information over ROS:

The Navigation Stack requires odometry information to relate the velocity of the mobile robot. So, the odometry source publishes the information about the velocity as well as uses tf (transform trees) to transform the wheels of the robot to the **base_link**. We have used ROS to publish odometry information through the **nav_msgs/Odometry** and also established a transform using tf to account for the changes in the coordinate frames.

The velocity control or the low-level control for the robot was done using Arduino Mega and this information was provided to the Jetson Nano which performed the computations and output the velocity commands back to the

Arduino Mega.

- Arduino software design:

```
#define EN_BL 45
#define IN1_BL 8
#define IN2_BL 9
#define EN_BR 46
#define IN1_BR 10
#define IN2_BR 11
#define EN_FL 12
#define IN1_FL 5
#define IN2_FL 4
#define EN_FR 13
#define IN1_FR 7
#define IN2_FR 6
```

```
#define IN2_FR 6
#define ENCODER_FR 3
#define ENCODER_FL 2
#define BUFFER_TIME 1000
#define ENCODER_BRA 19
#define ENCODER_BLA 18
#define ENCODER_BRB 20
#define ENCODER_BLB 21
#define PULSES_PER_REV 280
#define pi 3.14
```

This shows the pin initializations done in the Mega. Enable pins were used from the L298 motor driver and connected to PWM pins for speed control.

BL Back left motor

BR Back right motor

FL Front left motor

FR Front right motor

Encoder pins from the motors were utilized to obtain the pulses and the ticks per meter. For our use case, we needed both the channels (Channel A and Channel B) of the Encoder DC motors. So, we have used two channels from the back left and the back right encoded motors which is initialized as: **ENCODER_BLA, ENCODER_BLB** and **ENCODER_BRA, ENCODER_BRB** respectively.
Some other initializations for the hardware are as shown:

```
volatile long rotary_encoder_right=0;
volatile long rotary_encoder_left=0;

float left_rev_count = 0;
float right_rev_count = 0;
float l_dist_val=0.0,r_dist_val=0.0;
volatile boolean fired_left
,up_left,fired_right,up_right;
unsigned long current_time = 0,previous_time = 0;
//x,y wheel odom positions

//Robot Parameters
float wheel_rad = 0.030;
float w_fr;
float w_fl;
float w_bl;
float w_br;
float wheel_sep = 0.175;

//const float dist_per_rev = (2 * pi * wheel_rad) * PULSES_PER_REV;
const float dist_per_rev = (2 * pi * wheel_rad);
```

The wheel radius was found to be 3cm(0.030m) and the separation between the wheels i.e., the width of the robot was 17.5cm. The **w_fr, w_fl, w_bl, w_br** indicate the velocities of the respective wheels.
The distance covered by each wheel would be the circumference of the wheel given by: 2*π*r, where r is the wheel radius(**wheel_rad**). **l_dist** and **r_dist** are the respective distances covered by the left and the right motors.
Since we need to send the distance values to the processor for computation of velocities, we have used the concept of ROS publisher and messages which is shown in the next snippet:

```
// --------------ALL ROS MSGS----------------
std_msgs::Float32 l_dist_msg, r_dist_msg;   //gives use left wheel and right wheel distances pointers

//ROS Odom nodehandle and transform
ros::NodeHandle n;

// ROS PUBLISHERS
ros::Publisher l_dist("l_dist", &l_dist_msg);
ros::Publisher r_dist("r_dist", &r_dist_msg);
```

```
float speed_ang=0, speed_lin=0;

void cmd_vel_callback( const geometry_msgs::Twist& msg){
  speed_ang = msg.angular.z;
  speed_lin = msg.linear.x;
  w_fr = (speed_lin/wheel_rad)+ ((speed_ang*wheel_sep)/(2.0*wheel_rad));  // spee
  w_fl = (speed_lin/wheel_rad)- ((speed_ang*wheel_sep)/(2.0*wheel_rad)); //
  w_br = (speed_lin/wheel_rad)+ ((speed_ang*wheel_sep)/(2.0*wheel_rad));
  w_bl = (speed_lin/wheel_rad)- ((speed_ang*wheel_sep)/(2.0*wheel_rad));
}
ros::Subscriber<geometry_msgs::Twist> sub_cmd_vel("cmd_vel", &cmd_vel_callback );
```

cmd_vel_callback function was used to calculate the velocities of the wheels as shown in the snippet. We subscribed to **geometry_msgs/Twist** and obtained linear velocity x which is fed to **speed_lin** and the angular velocity was fed to **speed_ang**. The motion of the robot can be analyzed from the formula used. Consider the case of forward motion, where x and z values are 1 and 0 respectively, received from the subscribed message. Since **speed_angular** is 0, the second term in all the equations becomes zero and **w_fr, w_fl, w_br, w_bl** have a positive value. Similarly in case of backward movement the **speed_ang** and **speed_lin** values are 0 and -1 respectively which when evaluated results in **w_fr, w_fl, w_br, w_bl** equals to a negative constant value.

In case of left turn as explained in previous section, the right-side wheels of the robot are in forward motion and the left ones are in reverse. The **twist** message for this left turn is **speed_lin=0** and **speed_ang=0.5**. From the equations it is evident that the **w_fr** and **w_br** have a positive value and **w_fl** and **w_bl** have a negative value, leading to a left turn. The vice versa of the above computations results in the robot to turn right.

```
ros::Subscriber<geometry_msgs::Twist>sub_cmd_vel("cmd_vel",&cmd_vel_callbac
k );
```

This is the callback function to our **subscriber**; this takes a constant reference of a message as its argument.

```
void pinmode_setup()
{
 pinMode(ENCODER_BRA, INPUT_PULLUP);
 pinMode(ENCODER_BLA, INPUT_PULLUP);
 pinMode(ENCODER_BRB, INPUT_PULLUP);
 pinMode(ENCODER_BLB, INPUT_PULLUP);
}
```

The **pinmode_setup()** function configures the encoder pins to input pullup mode.

```
void pin_init()
{
 pinMode(EN_FL, OUTPUT);
 pinMode(EN_FR, OUTPUT);
 pinMode(EN_BL, OUTPUT);
 pinMode(EN_BR, OUTPUT);
 pinMode(IN1_FL, OUTPUT);
 pinMode(IN2_FL, OUTPUT);
 pinMode(IN1_FR, OUTPUT);
```

```
 pinMode(IN2_FR, OUTPUT);
 pinMode(IN1_BL, OUTPUT);
 pinMode(IN2_BL, OUTPUT);
 pinMode(IN1_BR, OUTPUT);
 pinMode(IN2_BR, OUTPUT);
 digitalWrite(EN_FL, LOW);
 digitalWrite(EN_FR, LOW);
 digitalWrite(EN_BL, LOW);
 digitalWrite(EN_BR, LOW);
 digitalWrite(IN1_FL, LOW);
 digitalWrite(IN2_FL, LOW);
 digitalWrite(IN1_FR, LOW);
 digitalWrite(IN2_FR, LOW);
 digitalWrite(IN1_BL, LOW);
 digitalWrite(IN2_BL, LOW);
 digitalWrite(IN1_BR, LOW);
 digitalWrite(IN2_BR, LOW);
}
```

The **pin_init()** function configures the encoder pins and motor connections to OUTPUT mode and initially sets to LOW.
The setup() function in turn calls the **pinmode_setup(), EncoderInit()** and **pin_Init().**

```
    n.initNode();
    n.advertise(l_dist);
    n.advertise(r_dist);
    n.subscribe(sub_cmd_vel);
```

The node handle is initiated and **l_dist** and **r_dist** are advertised creating a ROS publisher which is used to publish on a topic. Also the subscriber function is called.

```
void EncoderInit()
{
  attachInterrupt(digitalPinToInterrupt(ENCODER_BLA), isr_left, CHANGE);
  attachInterrupt(digitalPinToInterrupt(ENCODER_BRA), isr_right, CHANGE);
}
```

This function configures the back wheel encoder pins to Interrupt mode and attach the appropriate routine to it which is the **isr_left** and **isr_right** routines.
The interrupt routines update and the encoder counts.

The **update_left** and **update_right** functions returns the distance covered by each wheels which are required for wheel odometry and the values returned are then published by the **to_odom_function()**.

```
void MotorFL(int pwm);
void MotorBL(int pwm);
void MotorFR(int pwm);
void MotorBR(int pwm);
```

The above functions control the state of the motor input pins and the analog output based on the **pwm** value passed hence controlling the motor speed.

- Processor-side (Jetson Nano/Raspberry Pi 4) ROS design:

This part of the odometry computation was written in C++ using the attributes of **roscpp** which is provided as a dependency. Relevant header files were added to the software programs and codes.

```
class Encoder_Values {
    public:
        float left_enc_val=0.0;
        float right_enc_val=0.0;
        void left_enc_cb(const std_msgs::Float32::ConstPtr& left_msg);
        void right_enc_cb(const std_msgs::Float32::ConstPtr& right_msg);
        void reset(void);
};
```

Encoder_Values was a class we defined to specify the members of that class.
left_enc_cb and **right_enc_cb** were function declarations that were used to obtain the encoder data from the Arduino.

```
void Encoder_Values::left_enc_cb(const std_msgs::Float32::ConstPtr& left_msg){
   left_enc_val = left_msg->data; //left_distance_per_rev
//    ROS_INFO("left_enc_value: %f", left_enc_val);
}
void Encoder_Values::right_enc_cb(const std_msgs::Float32::ConstPtr& right_msg){
    right_enc_val = right_msg->data; //right_distance_per_rev
    // ROS_INFO("right_enc_value: %f", right_enc_val);
}
void Encoder_Values::reset(void)
{
    left_enc_val = 0.0;
    right_enc_val = 0.0;
    return;
}
```

```
int main(int argc, char **argv){
    float wheel_sep = 0.173;
    float distance = 0.0;
    float x = 0.0;
    float y = 0.0;
    float d_theta = 0.0;
    float theta = 0.0;

    float vx = 0.0;
    float vy = 0.0;
    float vyaw = 0.0;
```

Initializations of the parameters are done in this section. We have provided the **x, y** and theta(**th**) as 0.0 since the robot starts at the origin of the **"odom"** coordinate frame initially. The **d_theta** parameter is a differential amount which is used to find out the deviation in the robot's position.
The respective x, y and yaw velocities are also initialized to zero, so that the **base_link** will not move w.r.t the **odom** frame as soon as the robot is powered

up.

```
ros::init(argc, argv, "odom_cal");
ros::NodeHandle n;
ros::Time current_time, previous_time;

// ros variables used by logic
tf::TransformBroadcaster odom_broadcaster;      //used to update odom tf
nav_msgs::Odometry odom;                        //used to update odom values
Encoder_Values encoder_values;                  //class instance used to retrieve encoder distance
geometry_msgs::Quaternion odom_quat;            //used to get the quarternion/orientation for odom tf and odom
geometry_msgs::TransformStamped odom_trans;     //used to send the updated geometry messages to update odom tf
```

This handles the initializations on the ROS platform. **geometry_msgs**::**Quaternion odom_quat** is used to get the orientation of the odometry using Quaternions and publish this information via the **tf**. **geometry_msgs::TransformStamped odom_trans** is a transform message that is used to publish the transform from the "**odom**" frame to the "**base_link**" frame at **current_time**.

```
ros::Subscriber l_sub = n.subscribe("l_dist", 1000, &Encoder_Values::left_enc_cb, &encoder_values);
ros::Subscriber r_sub = n.subscribe("r_dist", 1000, &Encoder_Values::right_enc_cb, &encoder_values);
ros::Publisher odom_pub = n.advertise<nav_msgs::Odometry>("odom", 1000);
ros::Rate loop_rate(30);
```

These code lines above represent the subscriber and the publisher part that is important for transmitting the odometry information. This part shows the communication of data between the Arduino Mega and the main processor. The processor subscribes to **l_dist** and **r_dist** published from the Arduino and at the same time publishes a new message of the type **nav_msgs::Odometry** named as "**odom**". 1000 represents the **queue_size** for the number of messages or data that can be published and it was chosen so as to not cause overload for the buffer on the Arduino Mega. We are also setting a rate of 30Hz which maintains the loop at this rate.

```
while(ros::ok()){
    ros::spinOnce();

    current_time = ros::Time::now();

    // ROS_INFO("left_enc_value: %f", encoder_values.left_enc_val);
    // ROS_INFO("right_enc_value: %f", encoder_values.right_enc_val);

    distance = (encoder_values.left_enc_val+encoder_values.right_enc_val)/2;
    d_theta = (encoder_values.right_enc_val-encoder_values.left_enc_val)/wheel_sep;
    // resets encoder dist values for each motor
    encoder_values.reset();

    odom_quat = tf::createQuaternionMsgFromYaw(theta);

    // theta = theta*(180/pi);
    x = x + (distance*cos(d_theta));
    y = y + (distance*sin(d_theta));
    theta = theta + d_theta;

    vx = distance*cos(d_theta)/(current_time - previous_time).toSec();
    vy = distance*sin(d_theta)/(current_time - previous_time).toSec();
    vyaw = d_theta/(current_time - previous_time).toSec();
```

This loop is used to calculate the velocities in the x and y directions and also the rotational velocities. The distance travelled is usually calculated using the formula:



This formula was used, by giving the appropriate left encoder and right encoder values in the code.
Similarly, the deviation in the distance travelled by the robot was based on the following equation:



The distance travelled over time was incremented periodically by using basic trigonometric functions. The rotation angle value was also incremented by adding the deviation angle to the angle value.
Finally, the velocity was calculated using the well-known relationship between speed, distance and time:



Similarly, the change in the yaw angle or the rotational velocity was calculated by dividing the deviation angle by the time lapsed.

```
odom_trans.header.stamp = current_time;
odom_trans.header.frame_id = "odom";
odom_trans.child_frame_id = "base_link";
odom_trans.transform.translation.x = x;
odom_trans.transform.translation.y = y;
odom_trans.transform.translation.z = 0.0;
odom_trans.transform.rotation = odom_quat;

odom_broadcaster.sendTransform(odom_trans);

odom.header.stamp = current_time;
odom.header.frame_id = "odom";
odom.child_frame_id = "base_link";
odom.pose.pose.position.x = x;
odom.pose.pose.position.y = y;
odom.pose.pose.position.z = 0.0;
odom.pose.pose.orientation = tf::createQuaternionMsgFromRollPitchYaw(0,0,theta);
odom.twist.twist.linear.x = vx;
odom.twist.twist.linear.y = vy;
odom.twist.twist.linear.z = 0.0;
odom.twist.twist.angular.x = 0.0;
odom.twist.twist.angular.y = 0.0;
odom.twist.twist.angular.z = vyaw;

previous_time = current_time;

odom_pub.publish(odom);
loop_rate.sleep();
```

This part of the code is entirely oriented on ROS. We have published the transform from the **base_link** to the **base_camera** using "**odom_trans**" which is a **TransformStamped** message that publishes the **tf**. The parent or the header frame is the **odom** and the child frame is the **base_link**. The respective x and y transforms are provided and the rotational transform is published to the **odom_quat** parameter.
All these transformations are broadcasted through the **sendTransform** command.

The next part is used to fill in the transform message that is to be broadcasted by the **odom_trans** parameter. Similar to the transform structure, the parent and the child frame ids are provided and we are publishing the velocity information through the **nav_msgs/Odometry** message and the x, y and orientation of the pose is given. Through twist messages, the linear and the angular velocities are provided. We have considered the linear velocities in only the x and y directions and the angular velocities in the z direction. All these data are published through the **odom** parameter.

4. Mapping the environment and saving the map:
   As mentioned in Section 3.2 we are using Google Cartographer package to map the environment and used **teleop_twist_keyboard** package to move the robot manually in the room to cover all the areas. The **teleop** package directly gives velocity commands through the **/cmd_vel** topic to the Arduino which controls the motors of the robot. In addition, the laser scans were used to

detect the areas and visualize in **Rviz**. Through this function, we were able to map out any environment with a good accuracy and precision.
For saving the generated map, we used **map_server** package in ROS which provides the map data as a ROS service. **rosrun map_server map_saver -f map_name**: Using this command allows a dynamically generated map to be saved in .pgm format. It also generates a .**yaml** file which gives the description of the map.

```
image: /home/sumanth/catkin_ws/src/swarm_2dnav/map_home_handheld.pgm
resolution: 0.050000
origin: [-7.019227, -9.200000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

This is the **map.yaml** file which is generated by the program. It describes the resolution of the map (0.5) which means that each pixel can be converted to 5cm. Origin is the origin of the map which is the starting point from which the robot started obtaining the map. If the occupancy probability exceeds the **occupied_thresh** parameter, then it is represented as an occupied area and **free_thresh** parameter indicates the value of the free space in the bit map.

5. Launching the Navigation Stack:

At this stage, all the prerequisites/dependencies for launching the entire Navigation Stack were provided successfully. The main function of the launch file was to access all the dependencies necessary for Navigation Stack and bring up all the hardware dependencies and the transform links that are required for the robot.

There are two main parts to the launch file:

- Robot configuration file which launches the hardware dependencies such as the odometry source, sensor source (ZEDmini camera laser scans) and the transform configuration.
  This part of the launch file is shown in the below screenshot. The first package that is invoked is the odometry package: "**wheel_odometry**". Similarly, the **tf_broadcaster** is launched which brings the transform configuration for the robot. We have converted the output obtained from the ZED camera to laser scans in the next section. We are remapping from the topic: "**zed_node/depth/depth_registered**" where ZED publishes the depth info to the topic "**sensor_msgs/LaserScan**" and we are launching the "**zed_no_tf.launch**" to power up the ZED camera for visualization.
  In the next section, we are providing the map generated from map_server to the AMCL package for the purpose of localization. Since we are using an omni-directional robot, we have provided the parameter as "**omni**" for the model type. Some configuration parameters were provided for AMCL such as the minimum and the maximum particles to use for localization and the update rates respectively. The important part in configuring the AMCL package was to provide the transform frames provided for the odom and the

base frame of the robot as well as the global frame as "**odom**", "**base_link**" and the **"map"** respectively.

```
<launch>
 <node pkg="odom" type="wheel_odometry" name="odom" output="screen">
 </node>
 <node pkg="robot_setup_tf" type="tf_broadcaster" name="tf" output="screen">
 </node>
 <!--- Depth image to laser scan -->
 <node pkg="depthimage_to_laserscan" type="depthimage_to_laserscan" name="depthimage_to_laserscan" >
    <param name="scan_height" value="3"/>
    <param name="output_frame_id" value="base_link"/>
    <remap from="image" to="zed_node/depth/depth_registered" />
 </node>
 <!-- Zed Camera Configuration for PointCloud -->
 <include file="$(find swarm_2dnav)/launch/zed_no_tf.launch" />
 <!-- Run the map server -->
 <arg name="map_file" default="$(find swarm_2dnav)/map_home_handheld.yaml" />
 <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" respawn="true" />
 <!--- Run AMCL -->
 <!---include file="$(find swarm_2dnav)/launch/amcl_omni.launch" / -->
 <node pkg="amcl" type="amcl" name="amcl" output="screen">
   <remap from="scan" to="scan"/>
   <param name="odom_model_type" value="omni"/>
   <param name="odom_frame_id" value="odom"/>
   <param name="base_frame_id" value="base_link"/>
   <param name="update_min_d" value="0.1"/>
   <param name="update_min_a" value="0.2"/>
   <param name="min_particles" value="500"/>
   <param name="global_frame_id" value="map"/>
 </node>
```

- The second part of the launch file is used for launching the configuration files required for the operation of the **move_base** package. These are config files in **yaml** format that are used for setting parameters for the costmaps and planner algorithms.

```
  <!-- Path Planner and Controller nodes-->
  <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
     <rosparam file="$(find swarm_2dnav)/cfg/costmap_common_params.yaml" command="load" ns="global_costmap" />
     <rosparam file="$(find swarm_2dnav)/cfg/costmap_common_params.yaml" command="load" ns="local_costmap" />
     <rosparam file="$(find swarm_2dnav)/cfg/local_costmap_params.yaml" command="load" />
     <rosparam file="$(find swarm_2dnav)/cfg/global_costmap_params.yaml" command="load" />
     <rosparam file="$(find swarm_2dnav)/cfg/base_local_planner_params.yaml" command="load" />
  </node>
</launch>
```

The Navigation Stack uses two costmaps namely the global costmap and the local costmap for storing obstacle information. The global costmap is used for global path planning that is used for finding the path to the goal point and the local costmap is used for the local path planning which is used for immediate avoidance of obstacles and other disturbances in the environment. To use these costmaps and their corresponding planners, we had to set the optimal parameters for our custom robot. The following section specifies the description of each of these parameter files. All the units used in ROS are in SI units.

- **costmap_common_params.yaml:**
  This file is used to set some general parameters that are used by both the global and local costmaps for navigation.

```
robot_base_frame: base_link

max_obstacle_height: 2.0
#robot_radius: 0.2
footprint: [[0.152, -0.074], [0.152, 0.074], [-0.152, 0.074], [-0.152, -0.074]]
footprint_padding: 0.01

static_layer:
  map_topic: /map
  subscribe_to_updates: true

obstacle_layer:
  observation_sources: laser

  laser:
    topic: /scan
    sensor_frame: laser
    observation_persistence: 0.0
    expected_update_rate: 5.0
    data_type: LaserScan
    clearing: true
    marking: true
    max_obstacle_height: 2.0
    min_obstacle_height: 0.0
    obstacle_range: 0.25
    raytrace_range: 0.3

inflated_layer:
  inflation_radius: 0.15
  cost_scaling_factor: 3.0
```

The robot frame is set to **base_link** and the maximum obstacle height that can be detected by the camera was limited to 2m. This parameter file provides two observation sources: **laserscan** and **point_cloud**. Since we are using laser scans, we have considered the **laserscan** as our **observation_sources** and the next set of parameters are related to the observation source.
The **rostopic** where the laser scans are published is /scan topic which is of **LaserScan** datatype.

marking and **clearing** are two **Boolean** parameters which are set to true. Marking is used for the detection of obstacles and registering the obstacle information.
Clearing is used for clearing out this obstacle information from the sensor readings when the obstacle/object is no longer in the vicinity of the robot. The obstacle information should be updating continuously at a set frequency which is given by the **expected_update_rate** parameter for safe navigation of the robot.
obstacle_range is another important parameter which is set based on the dimensions of the map. It is used to limit the maximum sensor reading for registering an obstacle in the costmaps. Since our map of the environment was relatively small, we have kept the parameter accordingly as 25cm.
The **raytrace_range** parameter is utilized for clearing out free space in front of the robot as provided by the parameter. According to the parameter, the

navigation stack attempts to clear out free space so that the robot can try to navigate without getting stuck.
cost_scaling_factor is a crucial factor used for scaling the cost values of instances in the occupancy grid map. The formula for that is as follows:



Here, costmap_2d::INSCRIBED_INFLATED_OBSTACLE is 254. Since the equation is multiplied by a negative sign, and because of the exponent, the values for obstacles' cost changes drastically for a very small change in value.

The raw map displayed and used by the robot is not sufficient for the robot to safely navigate and it requires some adjustments and one such parameter used for this purpose is the **inflation_radius**.
This parameter can be set by the user depending on the use case and is very important for the purpose of navigation. It is the amount by which the map inflates the cost values of the boundary of the map and the obstacles. This is usually set so that the robot's center of mass represented by the coordinates of "base_link" does not collide with the obstacles or walls.

Other trivial parameters are present for visualization and debugging in Rviz. The **footprint** and **footprint_padding** is used to obtain a basic model in Rviz as shown by this screenshot:



Figure 3. 16: Robot footprint in Rviz

The green rectangular object is the visualization of the robot in Rviz.

- **global_costmap_params.yaml:**
  This file is used to set some parameters for only the global costmap in particular.

```
global_costmap:
  global_frame: map
  robot_base_frame: base_link

  update_frequency: 1.0
  publish_frequency: 2.0
  transform_tolerance: 1.0

  static_map: true
  height: 50
  widht: 50

  plugins:
    - {name: static_layer, type: "costmap_2d::StaticLayer"}
    - {name: inflated_layer, type: "costmap_2d::InflationLayer"}
```

transform_tolerance is measured in terms of seconds and represents the amount of duration to wait until all the transform trees between different frames are updated and then the **global_costmap** is updated. Since the number of coordinate frames for our use case were less, we have set a low tolerance. **update_frequency** parameter in Hz determines the frequency at which the **global_costmap** is updated in the loop.
publish_frequency (in Hz) is the rate at which the costmap publishes the visualized information from the sensor.
Since **global_costmap** is static (**static_map: true**) and does not change much over time, we have set these parameters to low values.
The width and height of the map in meters is provided and a corresponding resolution factor in meters/cell is provided so that the width and height are scaled appropriately.
The plugins are obtained from the ROS documentation and we have provided two such plugins for visualization purpose.

- **local_costmap_params.yaml:**

```
local_costmap:
  global_frame: odom
  robot_base_frame: base_link
  update_frequency: 1.0
  publish_frequency: 2.0
  transform_tolerance: 1.0

  static_map: false
  rolling_window: true
  width: 1
  height: 1
  resoultion: 0.05


  plugins:
    - {name: obstacle_layer, type: "costmap_2d::ObstacleLayer"}
    - {name: inflated_layer, type: "costmap_2d::InflationLayer"}
```

The parameters hold the same meaning as the **global_costmap** parameters except for a few changes. The **static_map** parameter is kept to false because the **local_costmap** is dynamic and keeps changing according to the robot's movement. Hence the rolling window is made true so that the **local_costmap** is fixated on the robot at all times.

- **base_local_planner.yaml:**

  After setting the parameters with the costmap, this file is used for the setting the planner parameters. Path planning in Navigation Stack uses two path planners, namely: local planner and a global planner. The global planner we have utilized is called **NavfnROS** provided by the stack and we have not changed the parameters w.r.t the global planner. In case of the **local_planner**, there are two planners, namely: **TrajectoryPlannerROS** and **DWAPlannerROS** which are subsets of the **base_local_planner**. This planner provides a custom controller to **move_base** so that the robot performs autonomous navigation.
  According to the official documentation from ROS, DWA path planner is usually used for large maps and navigating numerous doorways and corners and where huge precision and accuracy is required. But this requires more computational power. So, since our map is also relatively small, we have used **TrajectoryPlannerROS** which is sufficient for our use case and provided good results with reasonable accuracy.

```
TrajectoryPlannerROS:
  max_vel_x: 0.45
  min_vel_x: 0.1
  max_vel_theta: 1.0
  min_in_place_vel_theta: 0.4

  sim_time: 5.0
  sim_granularity: 1.0

  controller_frequency: 3.5

  acc_lim_theta: 3.2
  acc_lim_x: 2.5
  acc_lim_y: 2.5

  holonomic_robot: false
  meter_scoring: true
```

The above snippet shows the parameters used for the **TrajectoryPlanner**. The minimum and the maximum linear velocities are set considering the load on the robot and the friction exerted by the floor on the wheels of the robot. The minimum rotational velocity is also set and the **min_in_place_vel_theta** represents the rotational velocity with which the robot rotates at a particular point without any linear motion associated. Similar to the velocity parameters, acceleration values were also set based on default values from the documentation.

sim_time is the time taken for simulating forward trajectories in the subsequent time steps and **sim_granularity** is the step size (in m) to take in between waypoints in the planned trajectory.

controller_frequency is another parameter which was decided by trial-and-error and it represents the frequency (time lapse) at which the controller corrects the trajectory of the robot.

holonomic_robot is kept true because our wheels are Mecanum wheels which allow the robot to have more DOF and moves in directions not possible by ordinary robots.

meter_scoring is a Boolean parameter that is used to express the map units in meters or cells. So, we have kept this “true”.

All these configuration files were launched from the launch file shown above. This completes the setup and launching of the entire Navigation Stack on our custom robot. The robot was able to move and navigate around the environment with a small amount of deviation and the corresponding results are shown in Section 3.3

3.6 Control Law Implementation

3.6.1 Use Case Representation

For our demonstration, we have showcased:

- Task of pick-and-place by the robot in an environment.
- Considering the objects as sponges, the robot detects the sponge and obtains the distance of the object from the base of the robot.
- This distance is passed as a goal point to the Navigation Stack and this will make the robot move towards the object.
- When the robot reaches the object, a signal from the main processor is provided to the Arduino Mega which controls the gripper to grab the object.
- Once the object is picked, the final drop-off point is fed as the goal point to the Navigation Stack through which the robot finally drops the object at the goal point.

3.6.2 Control Algorithm

This section covers the flowcharts and the algorithms developed for our Control approach and the associated pseudo code.

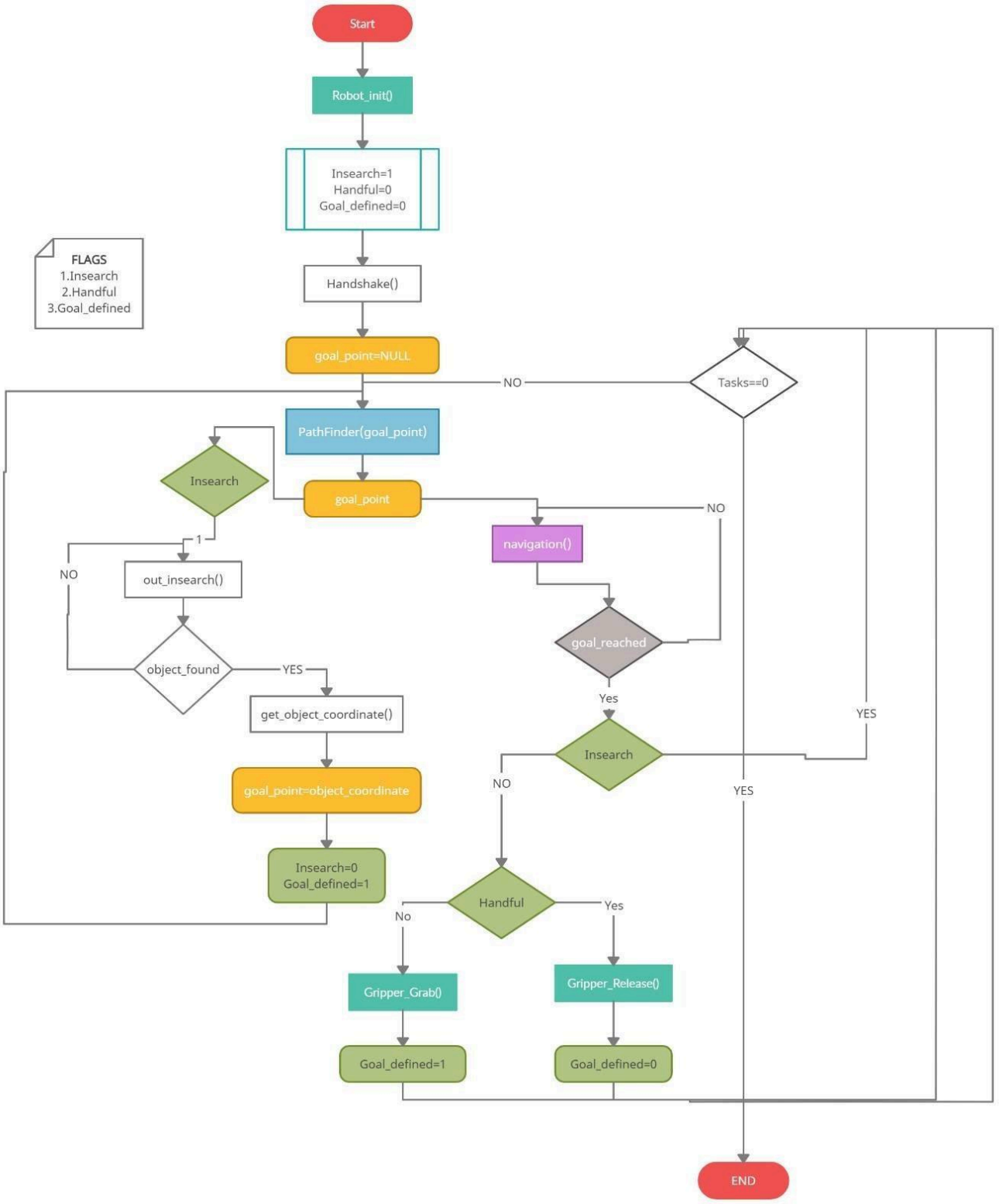


Figure 3. 17: Overall Swarm System Flowchart

The flowchart depicted above shows the overall working of the Swarm system.

This flowchart represents the working of the entire multi-robot coordinated system, handling all the test cases accounted for. The robots are first initiated where all the nodes are up and running and on standby waiting for the callback. There are 3 flags **Insearch**, **Handful** and **Goal_defined**.

Insearch flag is:

- **1** when the robots are in search of a package.

- **0** when the robots have already found a package i.e., when it's no longer searching.

Handful flag is:

- **1** when the robot is carrying the package
- **0** when the robot is not picking up or delivering a package.

Goal_defined is:

- **1** when the robot has been assigned a goal (this goal can be location of the package or the drop point and not any other intermediate goal).
- **0** when the robot has not received a goal point.

These three flags are used for the priority inspection depicted in the upcoming test cases.

The above-mentioned flags are preassigned with the values shown in the beginning. The **Handshake()** function establishes communication between the two robots, conducting a "ping" test to ensure stable communication link. Initially the robots do not have knowledge about the distribution of the packages in the environment and hence have no goal.
The **goal_point** variable is the parameter that is passed to the ROS Navigation Stack to travel towards that particular coordinate. It is a 2D vector containing the coordinate [*x, y*].
When the robot obtains the **goal_point**, the **PathFinder()** function gets executed and it provides this goal to the Navigation Stack. In other words, the function returns a **goal_point** value which will be used by other functions for navigation.

The **next_goal_point** is now the current **goal_point**. This value is now passed to the function **navigation()** which is responsible for robots' movement. The system will verify if the robot has reached the **goal_point** using **goal_reached** and if this Boolean value is false, **navigation()** function is revoked again till the robot navigates to **goal_point**.

In parallel to this process the **Insearch** flag is checked for its value and when it is **1** the node **out_insearch()** is started which executes the object detection program.
Once an object is found it breaks out of the loop and evaluates the object coordinates, using the **get_object_coordinates()**.
This new coordinate is then assigned to **goal_point** and is passed on to the **PathFinder()** so that the robot starts its movement towards this newly obtained coordinate. Meanwhile the flag **Insearch** is lowered by changing its value to **0**.
Once **goal_point** is reached, the flags are once again used to identify the situation.

- The **Insearch** flag is checked, if the value is found to be **1** then the robot has reached its intermediate goal point and then requests **PathFinder()** for the next **goal_point**.
- Otherwise, if the value is **0** then there is a possibility that it may have reached

a drop point or an object coordinate.

- To identify which case it is, the **Handful** flag is checked and if the vale is found to be **0** then the robot has reached an object and is ready for pickup.
- Conversely if the value is **1** then the robot has reached the drop point and drops the package.
- The **Gripper_Grab()** and **Gripper_release()** functions control the action of picking and releasing, respectively.

The variable **Tasks** keeps the count of total packages to be replaced in the warehouse and the count is decremented for every package dropped. Once all the tasks are over **Tasks=0** then the robot halts and that marks the end of the operation.

```
PathFinder goal_point =
        if              ==
                                =
                #the sine_path returns the next goal point as per the sine
        function such that the robot while searching, covers the entire area
        of the warehouse.
        elif type               ==
                        =
                # returns the current bot location

        else
                                =
        return
```

Through this process, the flags are utilized in an effective way to complete the coordinated task of pick-and-place operation by the autonomous robots.

The pseudo code for the entire algorithm is as shown below:

```
            list            #shared list for packet coordinate

dothemath   iamhere got_these =
        if          !=
            for         in
                if          not in
                                append
        else
            if          ==
                return

            = list
        for         in
                append sqrt

            #euclidean distance sqrt[(x2-x1)^2+(y2-y1)^2]
                =                   index min
                remove      index min
        return
```

The next flowchart is a subsystem flowchart representing the operation of the navigation part of the Control algorithm.

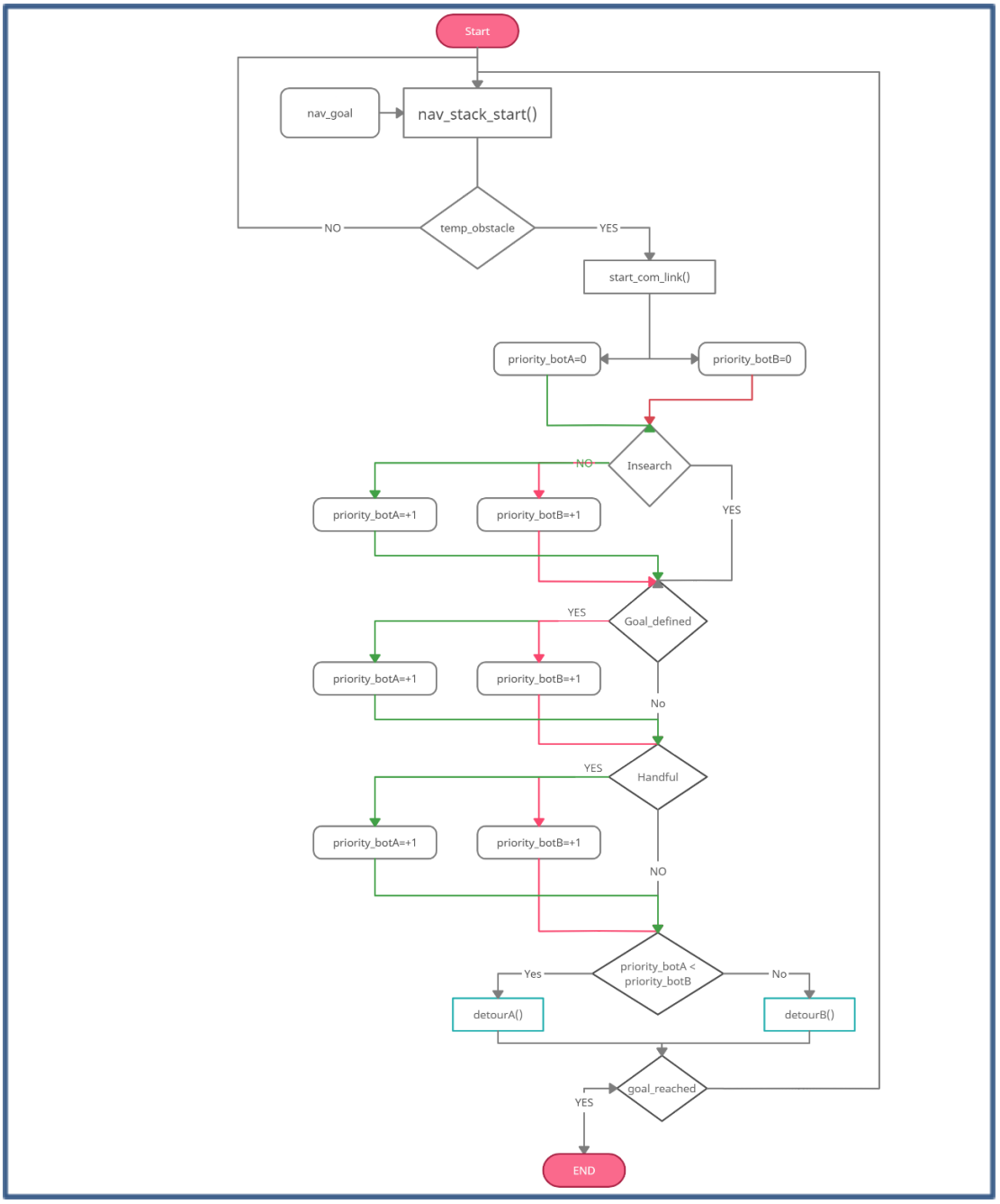


Figure 3. 18: Priority Scheduling algorithm

The **nav_stack_start()** handles all the ROS navigation stack nodes, which handles all the tasks with respect to navigation. The only modification occurs when it encounters a non-static obstacle, which for our case will be the other robot.

Once a temporary obstacle is detected, the robot starts communicating with the other robot, triggered using the function **start_com_link()**. Once both are communicating, they gather data from sensor and odometry streams and decide the priority among themselves. Initially both robots have a priority of **0** and subsequently, based on the flag values the priority is incremented. The following conditions are used for assigning the priority between the two robots for different scenarios and test cases.

- Prioritization:

- Robot with the package is given higher priority.

- When both robots have a package, the robot nearest to the goal point should be given higher priority.
- When both robots have no packages, the robot with pre-assigned priority will start to detect packages.

Note: As soon as temporary obstacles are detected, either robot starts communicating with the other about its status.

 Collision Avoidance:

The robot with higher priority will perform its operation while the other robot will be halted until it no longer detects the robot in its FOV or path.
The table below represents the different situations that might occur and the corresponding actions that are to be taken are illustrated.

Case	Status of Robot A	Status of Robot B
1.	No temporary obstacle	No temporary obstacle
2.	No temporary obstacle	Temporary Obstacle found
3.	Temporary Obstacle found	No temporary obstacle
4.	Temporary Obstacle found	Temporary Obstacle found

Table 3. 4: Temporary Obstacle Status

- For Case 1 the robots continue on their path without interruption.
- For Cases 2 and 3 the robots maintain their same path and when they are arriving at the destination, the robot that has detected the temporary obstacle will halt at some predetermined distance away from its destination for a certain time and then resumes its operation.
- For Case 4 the robots inspect priority and based on that, the robot with higher priority takes a detour and continues its operation and the other robot halts.

Based on these elementary rules and conditions, the Temporary Obstacle situation is handled and Navigation Stack will maintain the navigation of the robots until the goal point is reached.

4. RESULTS

This section demonstrates the working of the methods described in the previous section and the corresponding analysis of the results obtained.

4.1 Hardware Assembly and working

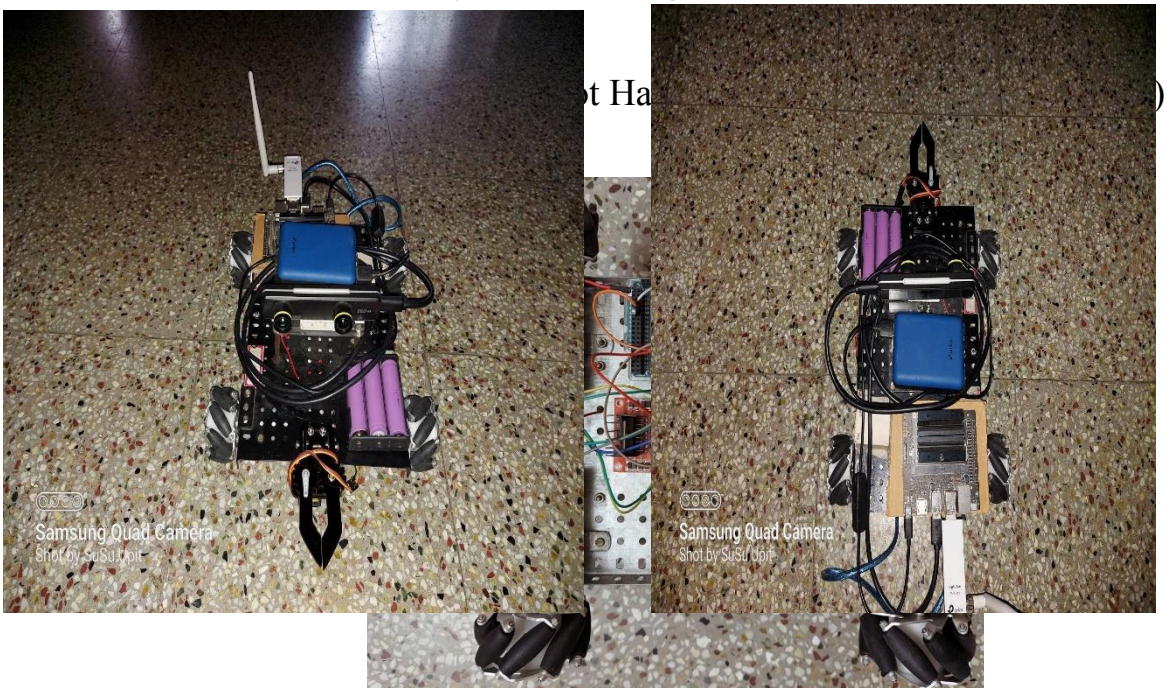


Figure 4. 2: Robot connections

One of our deliverables for the project was to develop the hardware for the two robots. As depicted by the above figure, we have integrated a gripper to the robot chassis and all the other components described in Section 3.1 are present onboard the robot.

4.2 Mapping

Maps generated by the Cartographer are quite accurate and were verified by repeating the mapping process for different environments. The screenshots for the different maps generated are shown below:

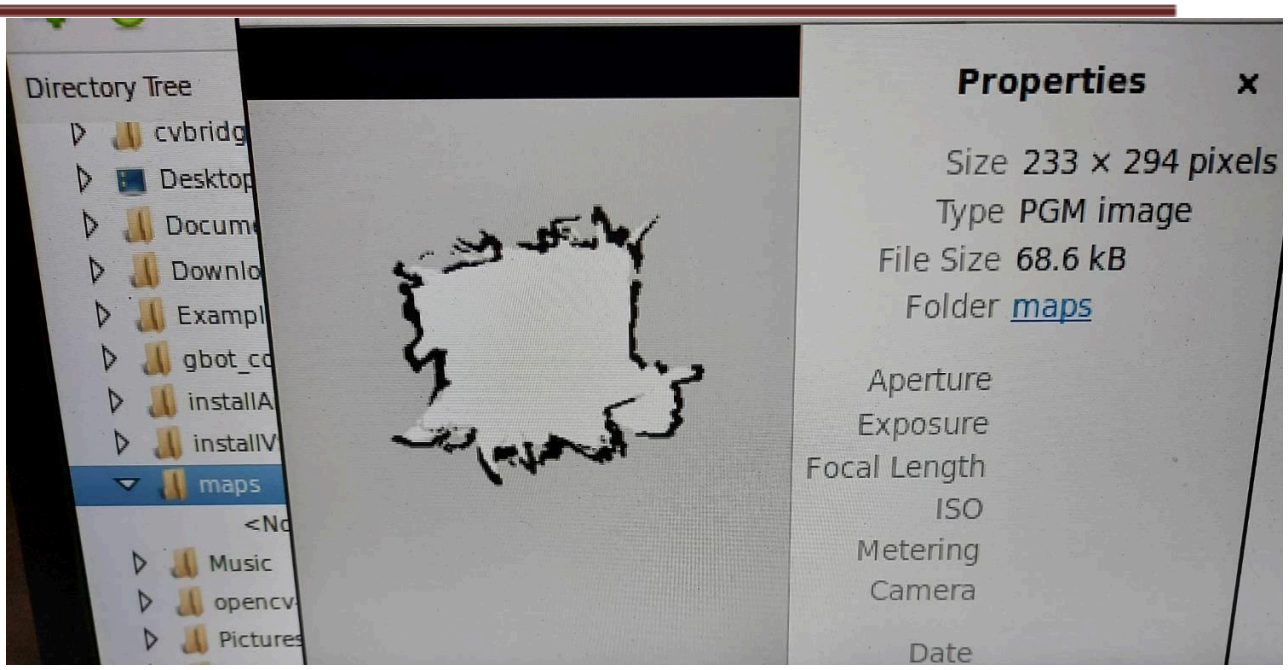


Figure 4. 3: Map (Occupancy grid) of a different room

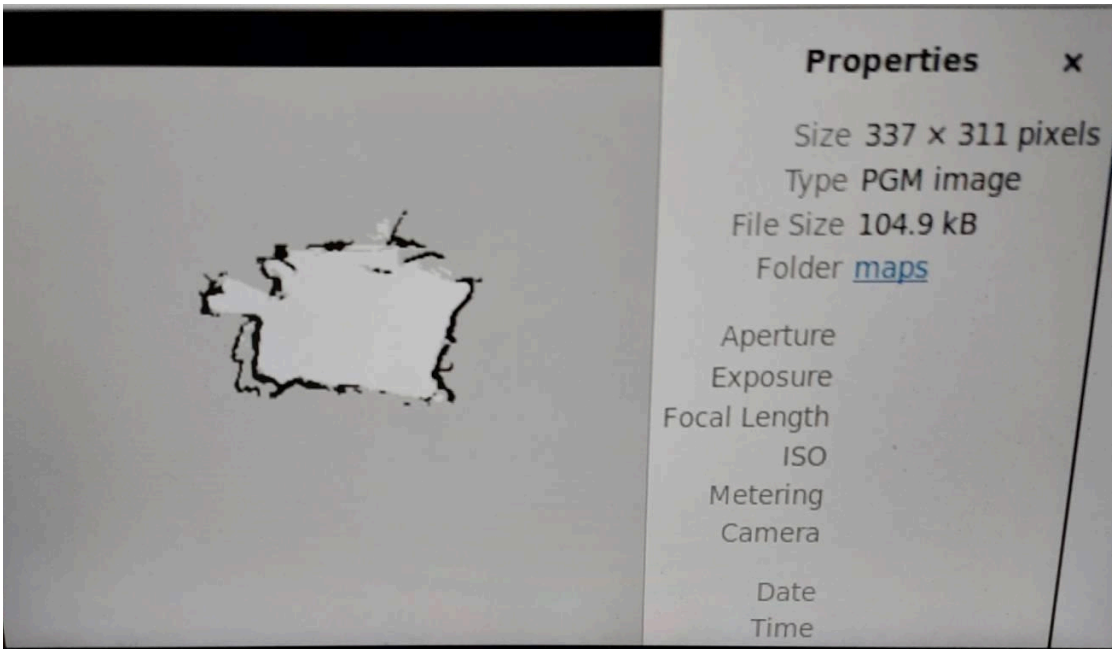


Figure 4. 4: Map (Occupancy grid) of the environment used

4.3 Localization

This table depicts the state estimation performed by our odometry code.
We have made our robot to move in different directions and measured the Pose values from our odometry.
The coordinates are in the form of (x, y), where x and y are in meters
The robot was made to move in a straight line in the positive X direction and made to move in reverse in the next iteration.
Other random movements were tested on the robot to validate the Odometry information.

Direction of movement	Ground truth pose value (x, y)	Estimated pose (x, y)	Pose Error = Ground truth – Estimated Pose
X-direction	(0.8,0)	(0.77,0.13)	(0.33, -0.13)
-X -direction	(0,0,0)	(0.07,0.12)	(-0.07, -0.12)
Strafing and reverse	(-0.2,0)	(-0.17,0.15)	(-0.03, -0.15)
In place rotation	(0,0)	(0.09,0.1)	(-0.09, -0.1)

Table 4. 1: Odometry State Estimation Table

From these results, we concluded that our odometry is associated with slip and there is error with state estimation. Hence, we are using AMCL localization based on depth camera scans which is used to provide a better navigation for the autonomous robot.

4.4 Computer Vision

The screenshot below shows the object being detected by the camera and applying the bounding box over the sponge.

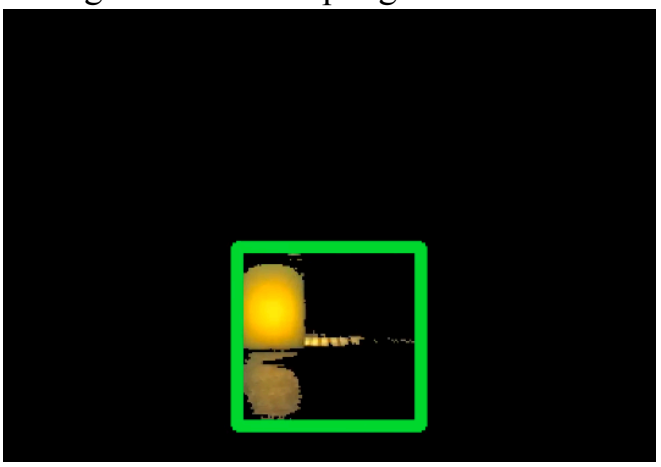


Figure 4. 5: Object detection by applying bounding box

The terminal shows the distance of the object from the camera which is shown in the below screenshot:

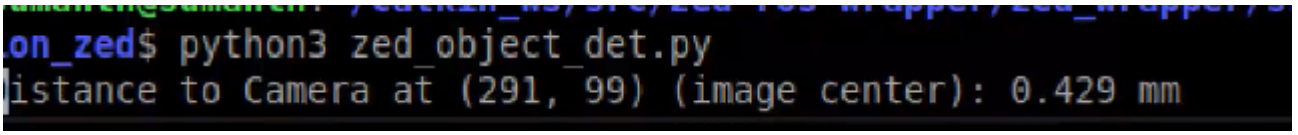


Figure 4. 6: Object distance shown in terminal

This was the output obtained when the object was placed too close to the camera.

The below observations depict the accuracy of the object detection by the ZED mini camera.
We tested the accuracy in a room with a sufficient amount of brightness and these were the results obtained for different test cases:

- Without computing the average value of the nearest 8 neighbors:

Object angle w.r.t camera	Reading #1(in m)	Reading #2(in m)	Reading #3(in m)	Average distance (in m)	Actual distance (in m)	Error (in m)
25^0	1.29	1.28	1.3	1.29	1.1	0.19
90^0	1.4	1.36	1.38	1.38	1	0.38
38.62^0	1.27	1.26	1.27	1.27	1.28	0.1

Table 4. 2: Accuracy table without averaging

- With computation of the average value of the nearest 8 neighbors:

Object angle w.r.t camera	Reading #1(in m)	Reading #2(in m)	Reading #3(in m)	Average distance (in m)	Actual distance (in m)	Error (in m)
25^0	1.093	1.091	1.093	1.092	1.1	0.008
90^0	1	0.985	0.991	0.992	1	0.008
38.62^0	1.23	1.22	1.2	1.216	1.28	0.064

Table 4. 3: Accuracy table with averaging

4.5 Navigation and Path-planning

The local costmap generated can be seen in the below screenshot:

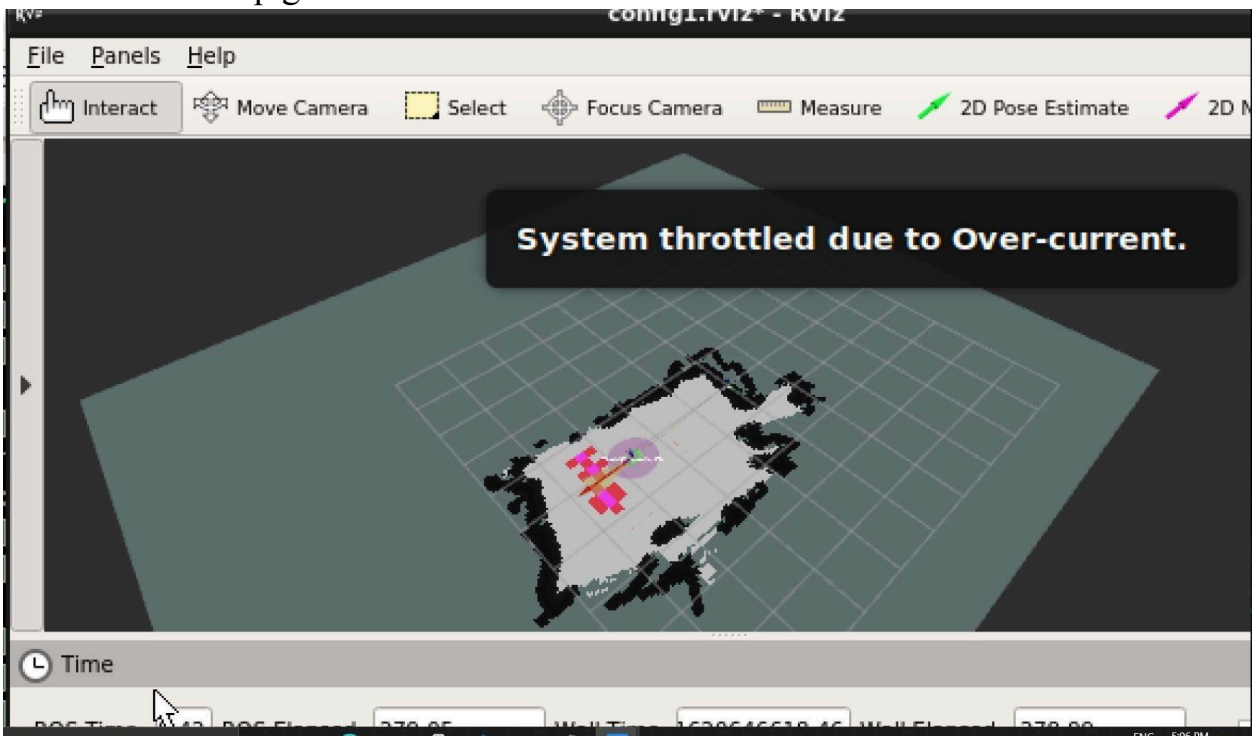


Figure 4. 7: Local Costmap visualization in Rviz

When an obstacle is present in the field of view of the camera, the local costmap gets updated automatically and this can be seen in the above screenshot as pink blocks.

4.5 CONCLUSIONS AND FUTURE SCOPE

At the end of Phase 2 we have successfully developed an autonomous robot which navigates around a known environment.

- Development of an application layer for Autonomous Navigation of a mobile robot by using ROS
- Completion of different modules for mapping, object detection and localization.
- Control algorithm design for task distribution and communication between robots.

We could not implement the Navigation Stack on our second robot due to the lack of a depth sensor and could not be acquired due to the present uncertain circumstances. Since the sensor stream is a mandatory prerequisite for ROS Navigation Stack, we could not make the second robot to perform autonomous navigation.

There is a lot of scope and scalability involved with a project of this magnitude. We have developed a completely scalable and portable Application Layer that can be used with any other robot with some minor changes with the parameters to suit the robot. With limited resources for computation, we have developed quite accurate modules, considering a trade-off between the main 3 parameters: cost, performance and power.
They can be replaced by more sophisticated algorithms with more advanced microprocessors to obtain more accurate and precise results for advanced use cases.

All the software components that have been developed in terms of our project have been documented and compiled on GitHub:
GitHub project Repository

Some developments that can be made to the existing modules are:

1. Use more advanced algorithms for localization (SLAM, RTABmap, etc.) and for path planning (RRT and RRT*, etc.).
2. Implement the designed Control Algorithm on the robots and test the feasibility, accuracy and overall working.
3. Scale the application layer to multiple robots and deploy the robots to perform real time tasks in any environment.

5 REFERENCES and LINKS

1. A Novel Swarm Robot Simulation Platform for Warehousing Logistics by Yandong Liu, Lujia Wang, and Cheng-Zhong Xu and Ming Liu: 2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)

2. Investigating the Democracy Behavior of Swarm Robots in the Case of a Best-of-n Selection by Yann Pochon, Rolf Dornberger, Vivienne Jia Zhong and Safak Korkut: 2018 IEEE Symposium Series on Computational Intelligence (SSCI)

3. B. Yamauchi, "A frontier-based approach for autonomous exploration," *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97*: Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. '

4. Depth Camera Based Indoor Mobile Robot Localization and Navigation by Joydeep Biswas and Manuela Veloso: 2012 IEEE International Conference on Robotics and Automation

5. Mauro, F.. “Towards the design of an effective and robust multi-robot parcel sorting system.” (2017): 2017 Thesis Paper TU DELFT

6. Ben-Ari, Mordechai & Mondada, Francesco. (2018). Elements of Robotics. Elements of Robotics. 10.1007/978-3-319-62533-1.

7. Springer Handbook of Robotics : Siciliano, Khatib (2008)

8. Planning Algorithms Textbook: Steven M.Lavalle

9. Sridharan, Anish; Patil, Vinay Venkanagoud; Bhat, Samarth; Venkatarangan, M J (2020). *[IEEE 2020 6th International Conference on Control, Automation and Robotics (ICCAR) - Singapore, Singapore (2020.4.20-2020.4.23)] 2020 6th International Conference on Control, Automation and Robotics (ICCAR) - Estimation and Control of Nodes in an Abstract Space.*

10. ROS Navigation Stack setup
11. ROS Navigation Stack setup
12. ROS Navigation
13. Setting up transform trees(tf)
14. Publishing odometry sources over ROS
15. Publishing sensor streams over ROS
16. Sending goals to the Navigation Stack
17. costmap_2D ROS Documentation
18. Navigation Stack Tuning guide
19. Navigation Stack Troubleshooting
20. map_server package wiki ROS page
21. tf Documentation
22. AMCL ROS Package

Swarm robotics

ORIGINALITY REPORT

8%	5%	4%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	wiki.ros.org Internet Source	1%
2	citeseerx.ist.psu.edu Internet Source	1%
3	Submitted to Midlands State University Student Paper	1%
4	www.stord.com Internet Source	1%
5	Spyros G. Tzafestas. "Mobile Robot Control and Navigation: A Global Overview", Journal of Intelligent & Robotic Systems, 2018 Publication	<1%
6	Studies in Computational Intelligence, 2016. Publication	<1%
7	Springer Handbook of Robotics, 2016. Publication	<1%
8	Ayssam Elkady, Tarek Sobh. "Chapter 13 Web-Based Control of Mobile Manipulation	<1%

	Platforms via Sensor Fusion", Springer Science and Business Media LLC, 2009 Publication	
9	Markus Hannebauer, Jan Wendler, Enrico Pagello. "Balancing Reactivity and Social Deliberation in Multi-Agent Systems", Springer Science and Business Media LLC, 2001 Publication	<1 %
10	Submitted to Universiteit van Amsterdam Student Paper	<1 %
11	hal.archives-ouvertes.fr Internet Source	<1 %
12	"Intelligent Robotics and Applications", Springer Science and Business Media LLC, 2019 Publication	<1 %
13	Submitted to Waikato Institute of Technology Student Paper	<1 %
14	link.springer.com Internet Source	<1 %
15	studentsrepo.um.edu.my Internet Source	<1 %
16	www.semanticscholar.org Internet Source	<1 %
17	www.coursehero.com Internet Source	<1 %

18	Submitted to Hull College, Humberside Student Paper	<1 %
19	Submitted to University of Surrey Student Paper	<1 %
20	www.theseus.fi Internet Source	<1 %
21	Submitted to University of Nottingham Student Paper	<1 %
22	Chia-How Lin, Kai-Tai Song, G.T. Anderson. "Agent-Based Robot Control Design for Multi-Robot Cooperation", 2005 IEEE International Conference on Systems, Man and Cybernetics, 2005 Publication	<1 %
23	Submitted to University of Sheffield Student Paper	<1 %
24	gerner.agat.net Internet Source	<1 %
25	Submitted to Middlesex University Student Paper	<1 %
26	Submitted to University of South Florida Student Paper	<1 %
27	github.com Internet Source	<1 %
	Submitted to University of Liverpool	

28	Student Paper	<1%
29	introlab.github.io Internet Source	<1%
30	"Vehicles, Drivers, and Safety", Walter de Gruyter GmbH, 2020 Publication	<1%
31	www.cesnet.co.za Internet Source	<1%
32	Jiang, Yinlai, and Shuoyu Wang. "Directional intention identification for running control of an omni-directional walker", International Conference on Fuzzy Systems, 2010. Publication	<1%
33	Yinlai Jiang, Shuoyu Wang, Kenji Ishida, Yo Kobayashi, Masakatsu G. Fujie. "User directional intention identification for a walking support walker: Adaptation to individual differences with fuzzy learning", The 6th International Conference on Soft Computing and Intelligent Systems, and The 13th International Symposium on Advanced Intelligence Systems, 2012 Publication	<1%
34	"EFSA's Catalogue of support initiatives during the life - cycle of applications for regulated products", EFSA Supporting Publications, 2016	<1%

	Publication	
35	A Cheong, MWS Lau, E Foo, J Hedley, Ju Wen Bo. "Development of a Robotic Waiter System", IFAC-PapersOnLine, 2016 Publication	<1%
36	create.arduino.cc Internet Source	<1%
37	utpedia.utp.edu.my Internet Source	<1%
38	www.mirlabs.org Internet Source	<1%
39	K. T. D. S. De Silva, B. P. A. Cooray, J. I. Chinthaka, P. P. Kumara, S. J. Sooriyaarachchi. "Comparative Analysis of Octomap and RTABMap for Multi-robot Disaster Site Mapping", 2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer), 2018 Publication	<1%

Exclude quotes	On	Exclude matches	< 5 words
Exclude bibliography	On		