Y.A.R.M – **Y**ARM's **A**nother **R**andom **M**otion planner: Enabling Multiple Mobile Robots to Remove Debris in Industrial Environments

Ankur Jaysukh Prajapati

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Mechanical Engineering

University of Washington

2020

Committee:

Ashis Banerjee, Chair

Santosh Devasia

Sawyer B. Fuller

Xu Chen

Program Authorized to Offer Degree:

Mechanical Engineering

University of Washington

Abstract

Y.A.R.M – YARM's Another Random Motion planner:

Enabling Multiple Mobile Robots to Remove Debris in Industrial Environments

Ankur Jaysukh Prajapati

Chair of the Supervisory Committee:

Professor Ashis Banerjee

Department of Mechanical Engineering,

Department of Industrial & Systems Engineering

Waste Management is essential to the Manufacturing Industry, and debris is a side product of many different kinds of manufacturing processes. Nowadays, mobile robots, and stationary manipulators, with multiple degrees of freedom, are increasingly used to carry out a variety of complex tasks. In such systems, the manipulators carrying out machining tasks may encounter a variety of obstructions in their machining paths, such as human hands, debris, or other robots. Debris is currently cleaned up manually by humans, before or after a machining process is complete. This may pose risk to humans at times, or affect the manufacturing process itself. To tackle this problem, this thesis proposes a novel random path planning algorithm, YARM, to generate paths for mobile robots, to reach the debris affecting the manipulator. The thesis describes the algorithm, the framework built to run it, and discusses its performance, pertaining to successful

collision avoidance and planned paths, in experiments conducted on a simulated system consisting of a manipulator, mobile robots and debris.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am extremely grateful to my adviser Prof. Ashis Banerjee who provided me the opportunity to contribute to the research work done at BARC Lab at the University of Washington. The time spent on this research work has helped me grow technically, professionally and emotionally. I thank Prof. Santosh Devasia, Prof. Sawyer Fuller and Prof. Chen for serving on my committee and for sharing with me their knowledge in Robotics and Control Theory.

I would also like to thank fellow BARC lab member Cameron Devine for guiding me to get the HC10 Sander simulation running for my project. Finally, I am forever grateful to my friends and family for their constant support.

DEDICATION

To my father and mother, Jaysukh and Anita Prajapati

And, my brother, Dipen

1. INTRODUCTION

1.1 Background and Related Work

Waste management is becoming increasingly essential to the manufacturing industry. Especially in processes that involve material removal, machines generate debris as side-products, which need to be removed, collected and disposed in a timely manner, to get the machines ready for the next batch of machining. The generated debris can be of various sizes and shapes, ranging from particulate matter of micrometer sizes, to chunks that are several millimeters in length. When a part has finished machining, the machines are stopped, and such debris is cleared from the workspace using pneumatic blowers or vacuums before removing the part from the workspace. Sometimes debris generated during the machining process can get in the path of the cutting tool and affect the tool tip. To try and prevent this, the cutting tools are often bathed in coolant, which not only cools the machining zone, but also helps remove the debris from that zone. However, this doesn't completely remove debris from the environment; the debris lies scattered in the surroundings, which is cleaned up by human operators. Such removal, collection, and disposal of debris is a common practice in many industries. The task can be split into many smaller stages – moving towards the machine, stopping the machine, clearing the debris by pushing it into a collection area, bagging up the debris into containers, and then taking these bags to waste or recycling containers.

Industries are researching the use of robots for waste management or manipulation tasks in certain areas. For example, for space applications, cleaning up of space debris is becoming increasingly important, and with robotic manipulators attached to spacecraft, potential solutions have been proposed[1][2]. Similarly, robots are being employed in industrial waste recycling, for sorting tasks, making use of novel computer vision techniques to identify different classes of waste[3]. Robots are particularly useful in search and rescue operations in disaster-ravaged areas, or for nuclear decommissioning [4], which often poses a high risk for humans to explore. However, there is a large gap that needs to be filled in terms of using robots to completely automate debris removal in industrial environments.

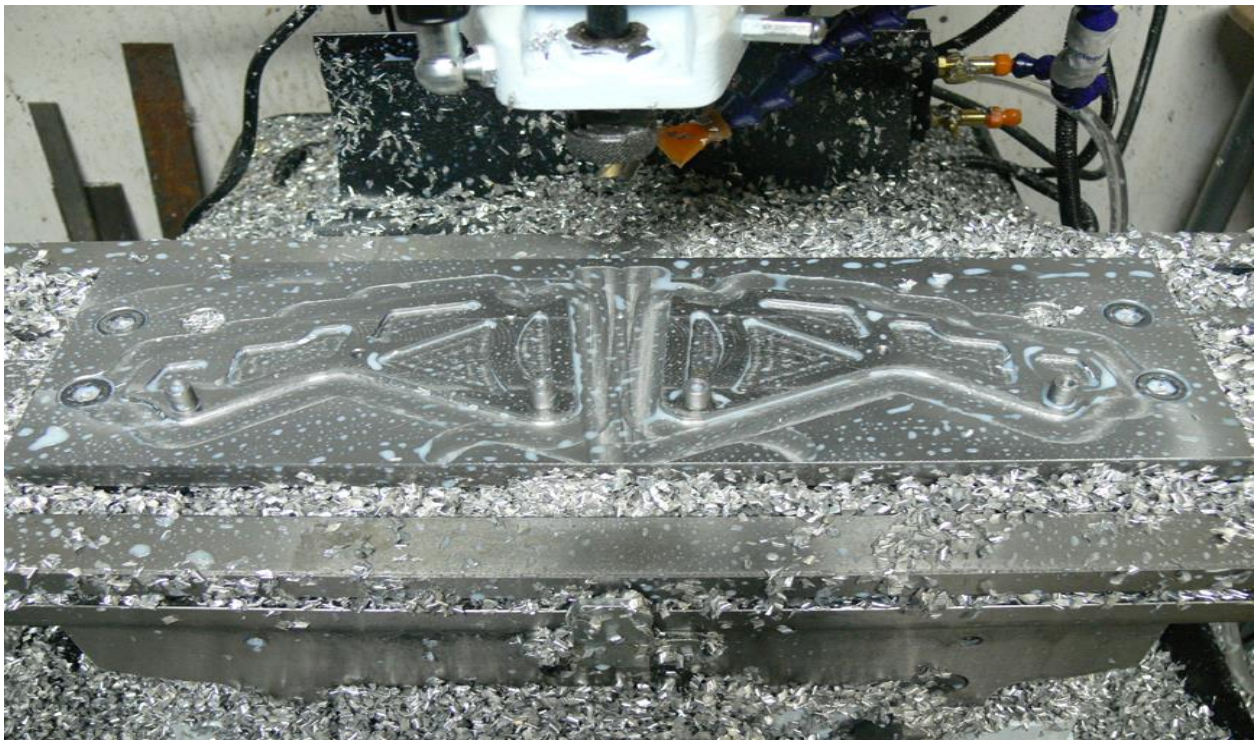
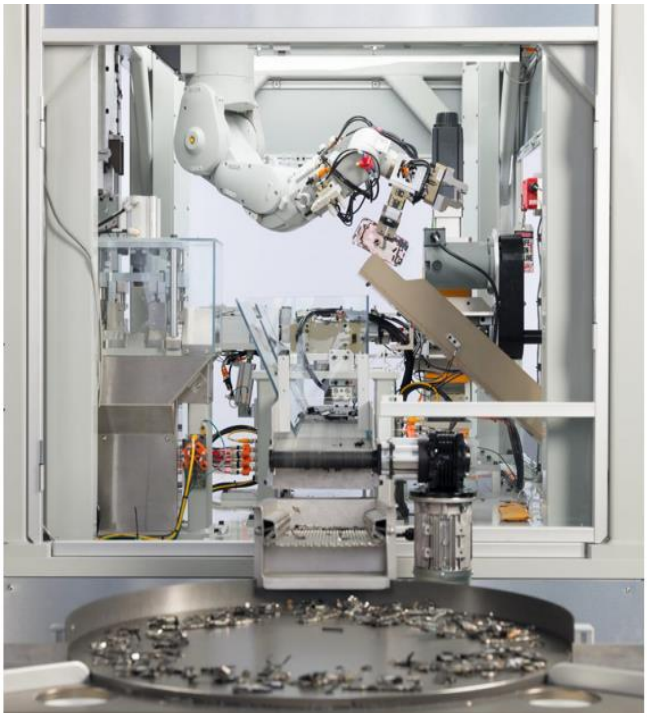


Figure 1.1: Metallic debris generated after completion of a milling operation on an aluminum block

Particularly relevant to the scope of this text is identifying debris and finding a path towards it, using an automated means such as a robot. Mobile robots have their applications in industry, for moving objects from point to point, which are often pre-programmed and labeled in the environment. A new generation of robots, such as the KUKA youBot[5] have arms attached to them for manipulation tasks, and also have a carriage to place items, so that such robots can be used for both, moving objects, as well as manipulating them. Such robots have high dimensional configurations, posing challenges to motion planning, such as simultaneously finding good paths and configurations for the robot at each step on that path. Searching the entire set of configurations for a 'best' configuration at each time-step is often infeasible, so such robots are programed to operate with a limited set of configurations. This allows the use of commonly available path-planning algorithms in a limited capacity. Often, the problem of simultaneous manipulation and motion by for robot's base in space, is decoupled from motion planning of its manipulator[6][7], which means that the mobile bases move to designated spots in the environment, and remain stationary there for the manipulator to take care of grasping tasks. Commonly used path-planning algorithms such as A*[8], RRT[9], JPS-A*[10], PRM[11] assume that the mobile platform robots

do not change shape during their motions, and thus can occupy a fixed set of cells which can be approximated by standard shapes like cubes, spheres or cylinders. A wide range of such motion planning algorithms are available in packages such as MoveIt![12]. These work extremely well for simple robot like Turtlebots, Husky AGVs, UR5 manipulators, that can be specialized for tasks. Some of these are randomized path planners; they use a Monte Carlo simulation strategy at some point in their process, to build possible paths, and their attractiveness for use is due to the fact that they are applicable to virtually any type of robots, and that they are probabilistically complete[18]. For stationary manipulators, sampling-based planning has also been researched actively[13][14][15], and incorporated in MoveIt!'s Open Motion Planning Library (OMPL). RRT has been implemented in MATLAB's Robotics toolbox[16] for use with stationary manipulators, for pick and place operations. Thus, there is extensive research on path planning algorithms for single stationary manipulators or single navigating mobile platforms, with readily available packages to use. Multi-agent systems pose additional constraints to the path-planning problem, and researchers have been trying to solve subsets of this problem by utilizing the strengths of known algorithms and simplifying state spaces so that the constraints don't pose challenges, or specializing the algorithms for tasks [17]. There is extensive literature on coordinated or cooperative motion planning for Unmanned Aerial Vehicles[19][20] and Unmanned Ground Vehicles[20], which make use of local and global planners to plan paths for short-term dynamism and collision avoidance, and obtaining a long-term start-to-goal solution. Such approaches of using local and global costmaps are quite common for path-planning algorithms. A* search is incorporated into the Gmapping[21][22] suite in the Robot Operating System (ROS) by making use of these methods, to deal with dynamism within a certain radius surrounding the robot.

As environments can get increasingly complex, so that the robot may have to change shape in order to fit through certain spots, common path-planning algorithms cannot, single-handedly, solve multi-agent coordinated path-planning problems. When trying to reach a debris, the robot base frame would have to compute 'best' configurations in the environment at each step that it moves, to avoid collisions, and also not have long wait times or extremely non-optimal paths, which decreases productivity. Decoupling the base and its on-board manipulator requires keeping track



(a)

(b)

Figure 1.2: Robots used for debris manipulation (a) A robot sorting components for recycling at Apple[3], (b) Multi-Arm Robot for Nuclear Decommissioning Tasks, (c) A trash-sorting robot designed by Alphabet's X Moonshot factory

of two frames. An alternative solution, then, as proposed by this thesis, is to plan paths between only the end-effector frame and the debris. This thesis is an attempt to couple motion planning for the two frames, given the knowledge that the manipulator is always fixed to the mobile base, and in doing so, provides an alternative solution to this specific class of high-dimensional problems by implementing a novel algorithm on a simplified representation of the high-dimensional mobile robot.

1.2 Motivation

Consider the interior panels of an aircraft. They are highly customized, contoured, complex part made up of composite materials, which are required to be light-weight, maintaining a good aesthetic as well as provide an additional layer of insulation to the aircraft interiors. These parts are required to be machined after removing from their molds, such as sanding them for better surface finish. At the BARC lab at the University of Washington, a Yaskawa Motoman HC10 manipulator is being used to research its applicability for the sanding operation[23]. Sanding is currently done by human operators in a closed room with vacuums to suction away the particulate debris, however, some particles may still remain floating in the air, and can be injurious to the operators if inhaled. Thus, a robot is better suited to carry out the sanding, and this extends to other kinds of operations as well such as drilling or grinding. However, human intervention is still required at the beginning and end of the machining process to clear the system of any debris, before loading and unloading the parts. Further, in the event that a human operator left behind ‘debris’ like loose bolts or nuts inadvertently, then the machining process has to be stopped entirely to avoid damage being done to the part or the machining process, and it may be risky for the human operator to intervene due to the particulate matter in the air[24]. As we progress onto more robots coordinating activities with humans in industrial environments, a mobile robot with manipulation capabilities is best suited to take care of such debris. It is to be noted, that debris removal need not be the primary task of such robots; the robots can carry out debris removal as a task in addition their currently assigned tasks. Intelligent automated scheduling systems can help with such coordination. Other than the example specified above, debris is generated while performing CNC milling, drilling and grinding operations, to name a few, and human operators need to clean the

workspace after the machining process is compete. The metallic debris can injure humans if not handled correctly, due to their sharp edges. Using robots for such clean-up tasks is a step towards completely automating waste management in the industry. If such a solution is scaled up and implemented, it will cut injury-related costs and improve human safety.

1.3 Contribution and Scope

The main contribution of this thesis is proposing the Y.A.R.M algorithm to move mobile robots, with attached manipulation capabilities, to debris positions, as a solution for removing industrial debris. Overhead cameras are used to generate a live map of the environment, as well as localize objects in the environment. A scheduler uses this information to plan paths for mobile robots to detected debris. The entire system will be controlled in a ROS (Robot Operating System) environment.

To help implement this algorithm in its current form, a simulated environment is setup in Gazebo Simulator, which handles the environment physics, as well as provides simple controller plugins for obtaining camera data, as well as controlling the robots. To limit the scope of this thesis, certain assumptions are made. First, it is assumed that the debris has fallen before the start of a machining process, the automated scheduler has detected the debris, and assigned the mobile robots the task of planning paths to their assigned debris. The mobile robots have an omnidirectional drive, so that they can move in 2D space, and have a simple cylindrical shape, but with a long extension ‘arm’ attached to them, which can be used to ‘reach’ or ‘touch’ objects in space. This is a simplification of a multi-DOF mobile manipulator, since such robots cannot be expressed as a single simple shape at all times, and therefore common path planning algorithms cannot be used for them. For this simple robot, currently only translation in space is explored; motion planning with rotation is not in scope. The debris is assumed to have constant shape and size at all times. The machining part is kept simple in shape, and the debris is loose, so it may move while on the part. For purposes of the experiments and easy visualization, the debris is assumed to take the shape of a door handle. Mobile robots are assumed to not retract or move away once they reach their assigned targets. The environment is assumed to be entirely indoors, which enables overhead cameras to facilitate capturing of the environment. Finally, the algorithm generates waypoints, and computations take place once all robots have reached their current waypoints, instead of

computation on-the-fly. Thus, certain elements of the environment and the solution are primitive, to provide a Proof of Concept, so that they can be used as reference for future improvements. As the path planning algorithm is the centerpiece of this thesis, tasks such as point cloud processing, or grid manipulation, have been done using standard libraries and packages, and a brief explanation of the underlying process, as well as references to the same, will be provided wherever applicable.

1.4 Organization of Thesis

This thesis comprises of two main parts, the first of which can be divided into four sections, and describes the setup of the simulation environment, and explains the pipelines that aid in providing inputs for the path planning algorithm to work on. Chapter 2 - "Simulation Environment" provides a top-level description of the ROS pipeline, its elements and their means of communication. Chapter 3 – "Computer Vision Pipeline" describes how the point clouds captured from the overhead cameras are converted into occupancy grids containing information about robots and obstacles in the environment. In Chapter 3, "Robot Controllers", the HC10 Sanding system's state machine and operation is described, and details of the modeled mobile robot are provided. In Chapter 4 – "Data Structures", the need for, and type of data structures used to handle various kinds of relevant data are discussed, along with an explanation of how they are related. This also gives an introduction to the state machine 'Commander' that would be the user interface for human operators overseeing the process. In the second part of the thesis, starting at Chapter 5 – "Path Planning Algorithm", the Y.A.R.M path planning algorithm is described in more detail, including how it makes use of the data structures explained in Chapter 4. In Chapter 6 - "Experiments and Results", details of experiments conducted, and observations, are provided. Images from the simulation are presented to show interesting features of the planning process. Finally, in Chapter 7 – "Future Scope", the limitations are addressed, and possible future work is discussed.

2. SIMULATION ENVIRONMENT

2.1 Operational Environment

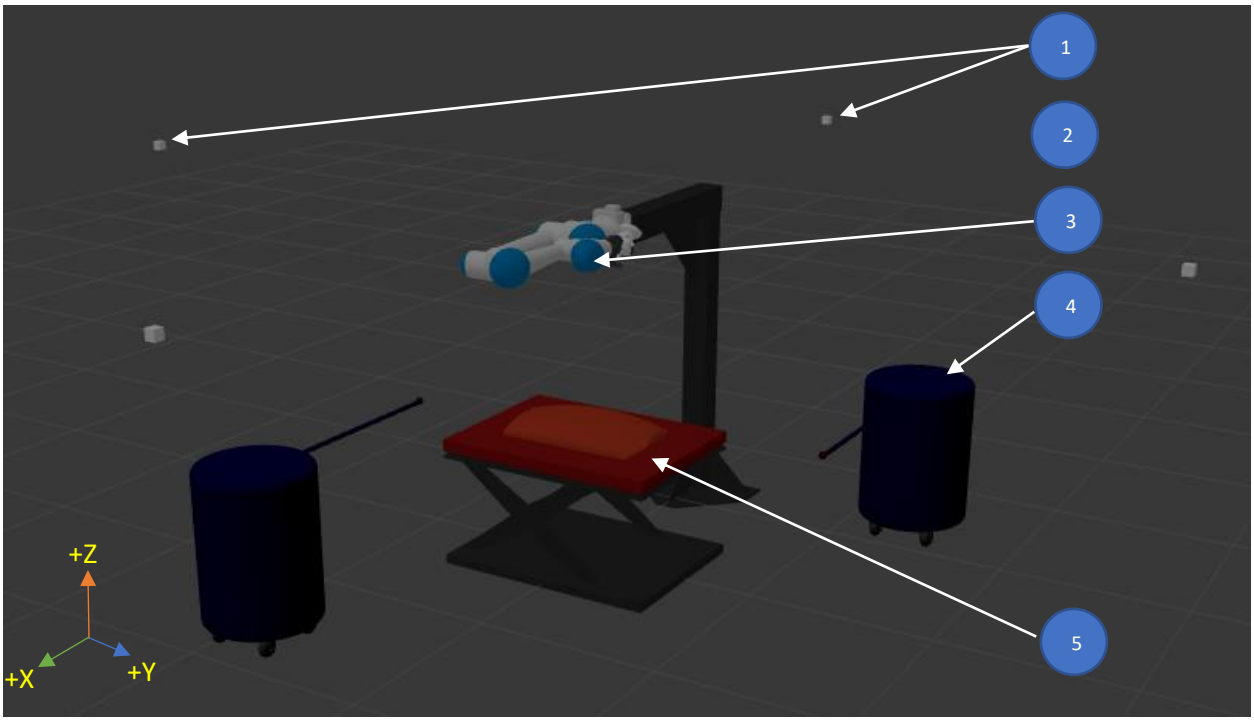


Figure 2.1: A screenshot of the environment as visualized in Gazebo Simulator – (1) Overhead Camera (Models represented as cubes), (2) Vertical Stand for Manipulator, (3) Yaskawa Motoman HC10 Manipulator, (4) Mobile Robot, (5) Table with Part to be machined

The environment used to test the algorithm simulates a subsection of an industrial workspace. This consists of the Yaskawa Motoman HC10 manipulator, its workbench and the part that it will be machining, as well as a portion of the total space. For the experiments, this space is restricted to [-2.5m, 2.5m] along X-Axis, [-2.5m, 2.5m] along Y-Axis and [0.01, 1.3m] along Z-axis. The axis directions are provided for reference in Figure 2.1. At BARC, the HC10 is placed inside a cage currently, however, this is to ensure no humans enter the cell during machining. Fortunately, as mobile robots are being used here, and no humans would be allowed on the premises, this

restriction can be relaxed. The size of the space also emulates situations where the robot is placed its own, larger, closed space, which mobile robots can freely enter and exit.

2.2 The Robot Operating System (ROS) pipeline

The Robot Operating System (ROS) [25] provides several tools to quickly setup prototype systems that can tested and later refined for production code. Thus, it provides an excellent means to set up the simulation from the ground up. For a path planning algorithm to function, it must be provided with information about the various objects in the environment. This is realized by means of a map (usually pre-made), as well as localization information about robots, which is generated by passing their sensor data through standard localization algorithms such as Extended Kalman Filters[27] or Particle Filters[26]. A limitation to using this methodology is that the environment must have been previously mapped using simultaneous localization and mapping (SLAM) techniques, and the maps needs to be continuously updated based on sensor information. Sensors need to be attached to every additional robot added to the system, and information fused from all the different streams of data from these sensors. The alternative proposed by this system is to have only basic collision-avoidance sensors, like ultrasound sensors, attached to the mobile robots. All identification and pose estimation (and therefore, localization) would be carried out using a Computer Vision pipeline. This enables the system to update the map in real-time, and accurately depict real-time dynamism. With the mapping and localization systems in place, data structures are needed to streamline data handling between various components of the system, as well as provide a user interface for humans supervising the operations from a distant point.

It is to be noted that identification of objects and their poses is not within scope of this thesis, as it is an extensive research topic in itself. Therefore, for the purposes of this simulation, the robot and object poses in this environment will be provided by directly referencing the ROS transforms tree, provided by tf and tf2_ros packages[28]. The descriptions of the robots are loaded as URDF (Unified Robot Description Format) files into the Gazebo Physics Simulator[30], which consists of a client and a server. The server keeps track of the objects in its environment and their interactions, and updates the transforms tree (tf) accordingly. Gazebo also provides simulated data from its Kinect plugins, which emulate a Kinect-like camera, providing Point Clouds of the scene. The point clouds and objects transforms are processed by a computer vision pipeline, which

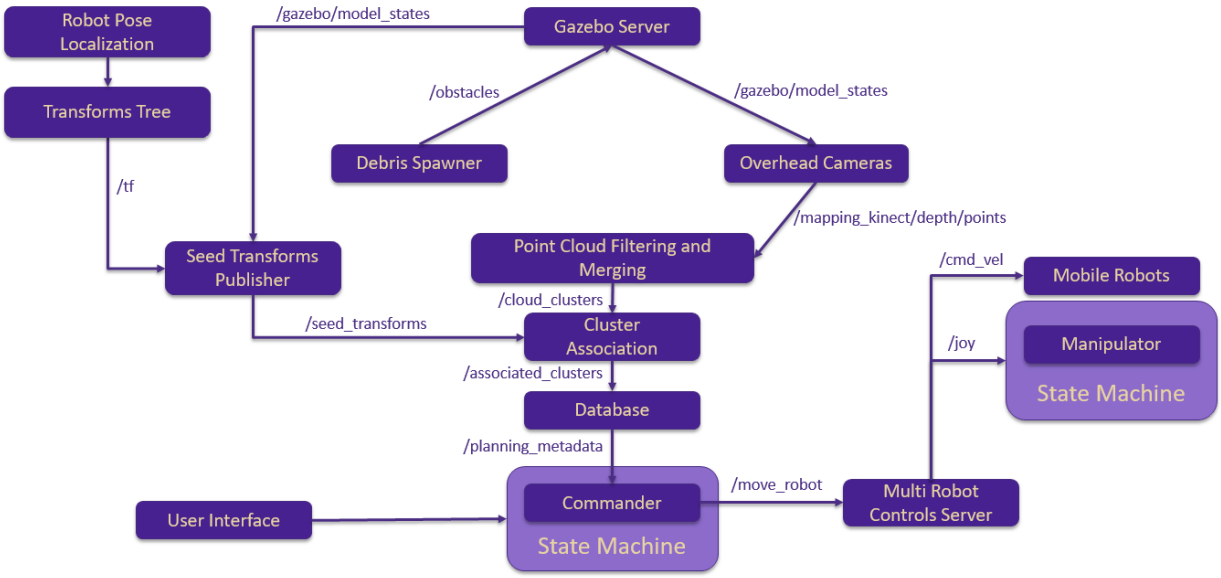


Figure 2.2: The ROS Pipeline, at the top-most level

associates point cloud clusters with the object that represents them. This data is stored in a database, which processes it into a 3D occupancy grid and generates metadata about the environment. The metadata is now available to a state machine, titled 'Commander', which provides an easy-to-use interface for human operators to set the state of the overall system. Commander provides a means to calibrate the entire system, as well has implements the YARM path planner during one of its states. Commands from the commander node are sent to a Multi Robot Controls server, which acts like a common interface for sending commands to many different kinds of robots – mobile robots and manipulators alike. Currently, the Multi Robot Controls server supports commands to the HC10 manipulator's state machine, and the mobile robot model that has been developed for this thesis. Figure 2.2 shows this ROS pipeline at the top level.

Visualizations of the environment are also made available using the RViz[30] visualization tool, available as part of ROS. The simulation currently runs on a laptop with Intel i7 2nd Generation dual core CPU with 8 GB RAM. The modules that perform various computations in this structure, called 'nodes' communicate via one of three provided modes within ROS: Messages, Services, and Actions. Messages ensure continuous streaming of data between nodes that publish data, and nodes that subscribe to that data. Messages are sent over 'topics', which are labeled in the pipeline above. Actions and Services work using clients and servers, so that data is only sent back by the

server when a client has sent a query to them. Similar to topics, they have their own two-way channels for sending and receiving data. As point clouds, robot transforms, and in turn, map metadata must be streamed continuously for real-time applications, they are sent via ROS messages. User interaction, and commands sent to the robots from the Commander, are implemented via ROS Services and Actions respectively. The Multi Robot Controls server receives ROS Actions from the Commander, and either streams ROS Messages or sends Service requests to the robot being actuated, depending on the robot's input interface.

3. COMPUTER VISION PIPELINE

3.1 Overview

The Computer Vision pipeline is used to generate map metadata, which informs the Commander node about the state of the environment. It makes use of several established algorithms, made available as part of the Point Cloud Library (PCL) [32], such as Euclidean Clustering, Box Filtering, and KDTree search. Figure 3.1 below shows the overall flow of this pipeline, and this will be followed by a brief discussion of computations occurring at each step, and parameters used for the same.

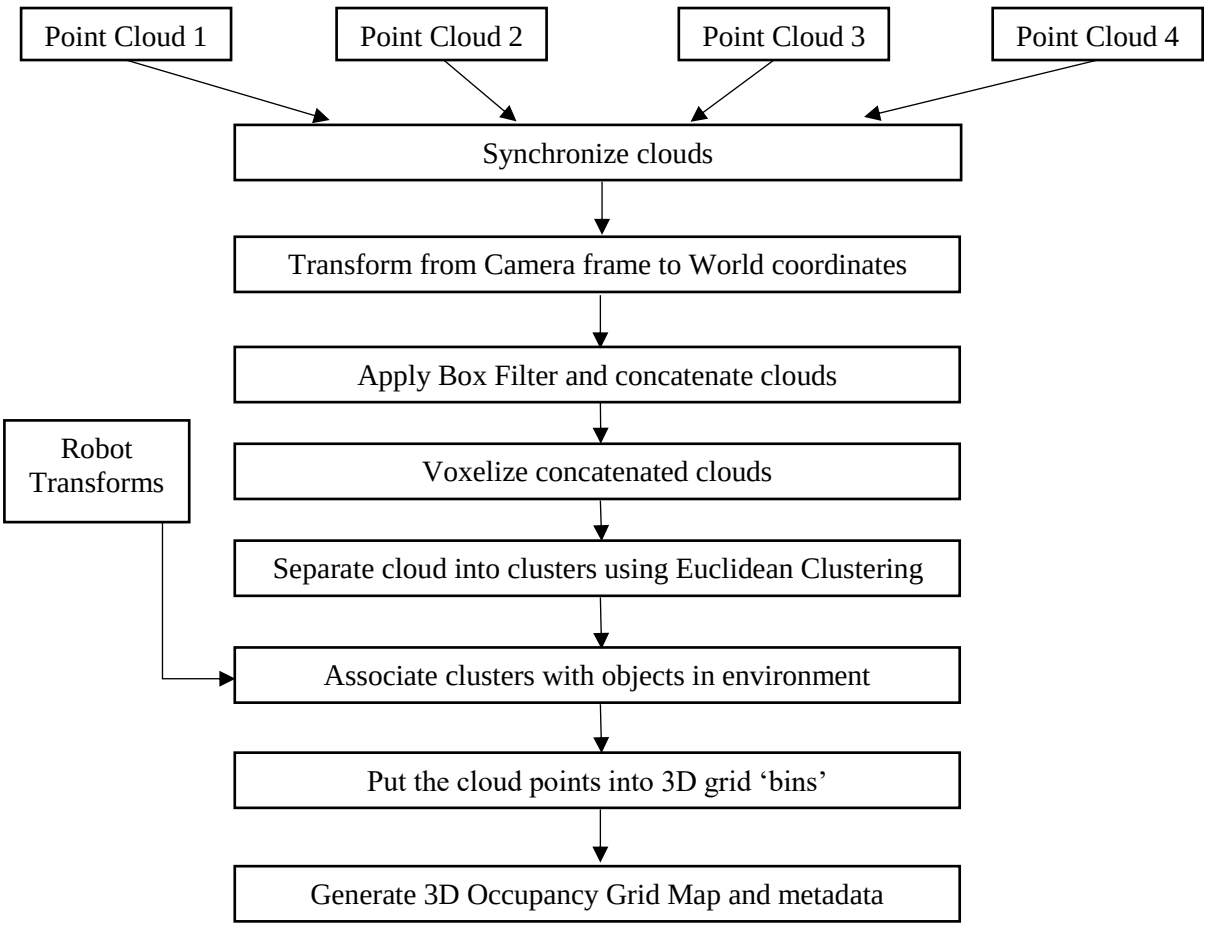


Figure 3.1: Process Flow of Computer Vision Pipeline

Point cloud data from the simulated environment is obtained from the Gazebo simulator. Gazebo provides the **libgazebo_ros_openni_kinect.so** plugin to emulate a Kinect camera. It offers some flexibility in adjusting the camera parameters, such as distortion coefficients, update rate and resolution. For the purposes of this simulation, the cameras have the following parameters:

Parameter	Value
Update Rate	30 FPS
Point Cloud Cutoff Minimum Range	0.5 m
Point Cloud Cutoff Maximum Range	10 m
Resolution	320 x 320 pixels
Horizontal Field Of View	135^{o}
Distortion coefficients K1, K2, K3, T1, T2	0.00000001

Table 3.1: Camera Parameters for Kinect Model simulated in Gazebo

The distortion coefficients are set to their default values, and as this thesis does not deal with camera calibration, these values were not altered. Further, this plugin emulates a general depth camera, so any camera that can provide point clouds with the above parameters, would work fine with the system. For reference, there are 4 overhead cameras located at the positions (3m, 0m), (0m, 3m), (0m, -3m) and (-3m, 0m) in the environment, at a constant height of 2m above ground level.

3.2 Processing the Point Clouds into Clusters

As point cloud processing is computationally expensive, there is a need to down-sample the received data into manageable chunks. The first step in this process is to ensure that data from all four overhead cameras are synchronized. This is done by making use of ROS Message

Filters and thread mutexes, which ensure that further processing takes place only once all clouds at one time step are received. The point clouds are sampled at a rate of 1000 Hz, then transformed from the camera frame to the world frame, using TF's **lookupTransform()** and **doTransform()** methods. The lookupTransform() looks up the camera's current transform with respect to the world, and transforms the point clouds using the relative transforms between the two. Next, the transformed clouds are limited to fit the range of [-2.5, 2.5] meters along X and Y axes, and [0.01, 1.3] meters along Z axis, using PCL's Box Filter function. The clouds are then concatenated and voxelized using PCL's VoxelGrid filter, with parameters [0.1, 0.1, 0.001] along X,Y and Z directions respectively. Voxelization downsamples points to one point per grid cell of a theoretical grid in space. This is done by finding the centroid of all points that belong to a certain cell in that theoretical grid, and replacing all points with that centroid. Finally, the voxelized cloud is clustered using PCL's Euclidean Clustering functions, using cluster tolerance of 2cm, minimum and maximum cluster sizes of 5 and 25,000 respectively. Euclidean clustering works by using random points from the cloud as 'seeds', and associating them with neighbouring cloud points based on a distance metric, the cluster tolerance. Finally, the clusters are sent to another node, a 'cluster association node' for associating with object transforms.

Point clouds, as per the PCL library, are stored in a KDTree data structure format, which allows quick, optimized lookup of points. Thus, when the clusters are received by the cluster association node, their KDTrees are stored in memory. The transforms of objects in the environment are treated as 'Seeds'. Since any manipulation task is done using the end-effectors, the transforms of the end-effectors are taken as seeds for the robots. For other objects in the environment, the seeds would be the centroid, or a point on the surface that is visibly connected to other points in its vicinity, for larger objects.

3.3 Cluster Association

Cluster association is tricky, because of the chances of bodies coming into contact. Often, clusters belonging to two bodies in close proximity to each other, or in contact with each other, coalesce into a larger cluster, and it is difficult to separate the two. Research has been going on in the field of computer vision[32][33] to done to effectively segment such contacting

clusters using 3D segmentation techniques. PCL offers an implementation of RANSAC that works well for cuboids, planes, cylinders and sphere, but does not work for complicated shapes, and thus, a better approach was needed to solve the problem of agglomerated clusters due to contacting bodies.

The method used here, stores point clouds of objects in the environment, during a 'calibration' phase of the Commander state machine. When calibration is being done, it is required that non-essential items not be in the workspace, and that no two objects are touching each other. The 'seeds' contain information such as their transform in space, name, type (Mobile Robot, Manipulator, Target, Debris, Obstacle), and whether or not the object is Static or Dynamic. By default, only stationary obstacles are Static. So, the vertical stand for the manipulator, and the table top with the workpiece, are Static. All other objects in the environment, including debris, are labeled Dynamic. The algorithm, then, goes as follows:

Cluster Association Algorithm

```
1) List_of_seeds = [Sd], [Sc]; List_of_clusters = [C], Pairs = []
2) If static_association_switch = False
      a. Save seed's current transform
3) Else if static_association_switch = True and seed_type = "Static
   Object"
      a. Do nothing
4) Else if static_association_switch = True and seed_type = "Dynamic
   Object"
      a. Save currently stored transform into a temporary variable
      b. Update the new transform
5) If static_association_switch = False
      a. For each static seed Ss, and cluster C
           i.  If rSearch(Ss, C) > 0
                    Add (Ss,C) to Pairs
      b. For each dynamic seed Sd,
           i.  If rSearch(Sd,C) > 0
                    Add (Sd,C) to Pairs
```

```
6) Else
      a. For each static seed Ss
           i. Transform saved associated cluster C to current
              transform of Ss
      b. For each dynamic seed Sd and new clusters C'
           i. If rSearch(Sd, C') > 0
                    Update (Sd, C') in Pairs
      c. For each dynamic seed Sd and its associated cluster C'
           i. Is rSearch(Sd, C'') > 0 where C' != C?
                 1. If yes, transform C' to current transform of Sd
                 2. Subtract C' from C''
                 3. Update any Sd' that is associated with C''
          ii. If C' is None
                    Save (Sd, C') to Pairs
7) Return Pairs
```

Here, *rSearch(S,C)* is a method that takes a seed point, and performs a radius search on the cluster, to check if there are any points from the cluster within radial distance of that seed. The method makes use of PCL's radiusSearch() method, that takes in a search point, search radius, and two empty vectors that it fills with neighbouring points and their distances from the search point, if found. Thus, if there are neighbours found for a given search point, the vectors would be of non-zero length, and this length is returned by the *rSearch(S,C)* function. The radius value for rSearch worked best with a value of 0.2m. The assumption that no two bodies are in contact during calibration, is a reasonable one, since calibration would be carried out only once per day or as per maintenance schedule. The seeds and their associated point cloud clusters are then sent onward to the next node for further processing. A visualization of the seed transforms and their associated clusters are shown in Figure 3.2.

3.4 Cluster Binning and 3D Occupancy Grid generation

A ‘database’ node receives the pairs of seeds and their associated clusters, and generates a 3D occupancy grid from the data. While not all tasks are mandatory, code for these tasks has been written to impart additional functionality that may be useful in future.

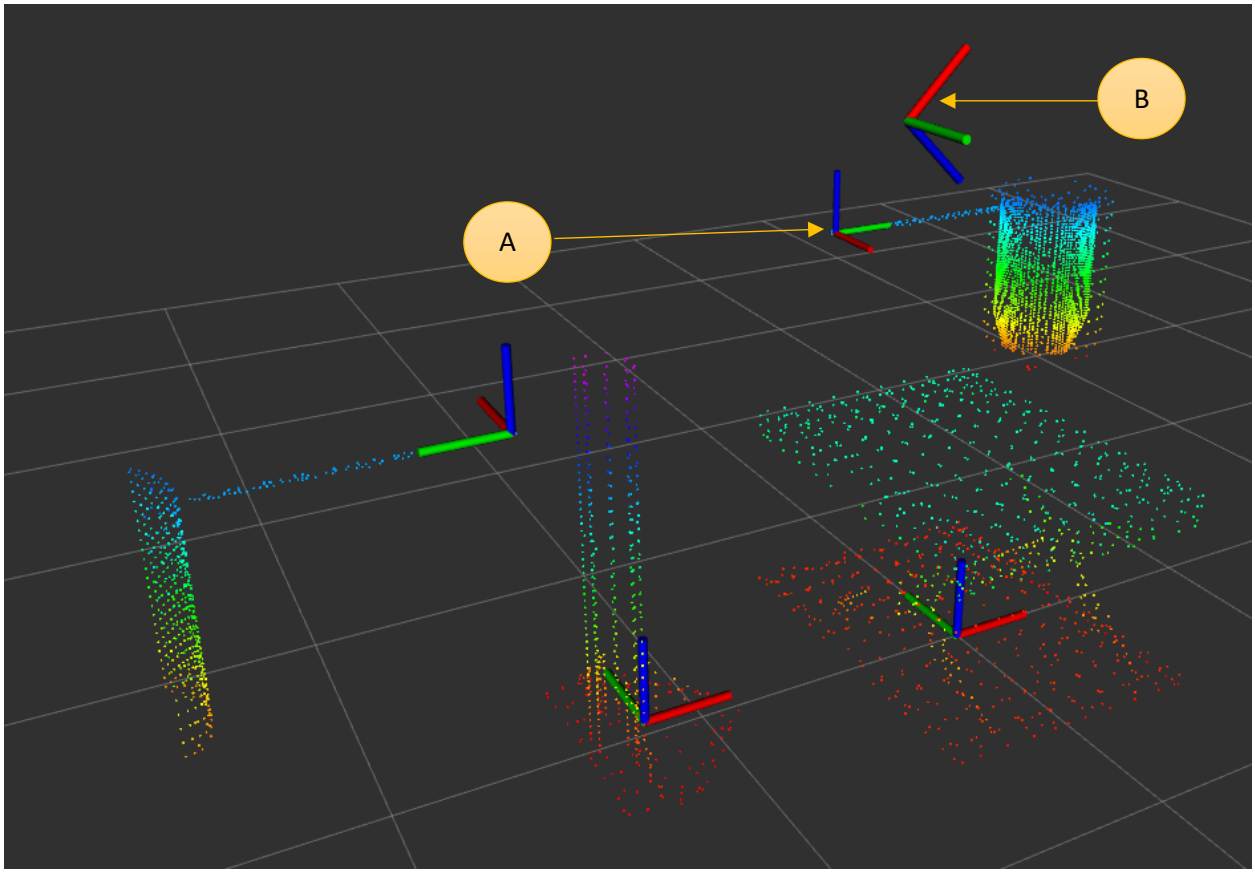


Figure 3.2: Cluster-Seed association – (a) Seed for Mobile Robot, associated with its cluster, (b) Seed for HC10’s end-effector, currently not associated with any cluster

A grid size of 50 x 50 x 20 (representing the X, Y and Z axes, respectively) is chosen, as it best represents the environment given the sparseness of the voxelized clouds before clustering, without having large gaps between grid points. Such gaps could potentially cause false

negatives during collision-checking. For the binning step, each point cloud cluster is iterated through, and the points are assigned to a grid cell. In doing so, some grid cells remain empty, some contain point clouds from only one object, and some may contain point clouds from many objects. The latter case is one to be careful about, for such objects would be at high risk of collision or contact. The grid is then passed through a filter that assigns occupancies: 0 for occupied by no object, 1 for occupancy by at most one object, and 2 for occupancies by multiple objects. This can be visualized in the example shown in Figure 3.3. This 3D occupancy grid structure, along with information such as seed transforms, object types, grid sizes, and the dimensions of the environment, are packaged into metadata, which is sent for use by the Commander. The database node also has a method, (currently not implemented, as it has no use for this thesis, since collision-checking is done by YARM) that parses the clusters and checks for collisions using a standard collision checker like GJK[34].

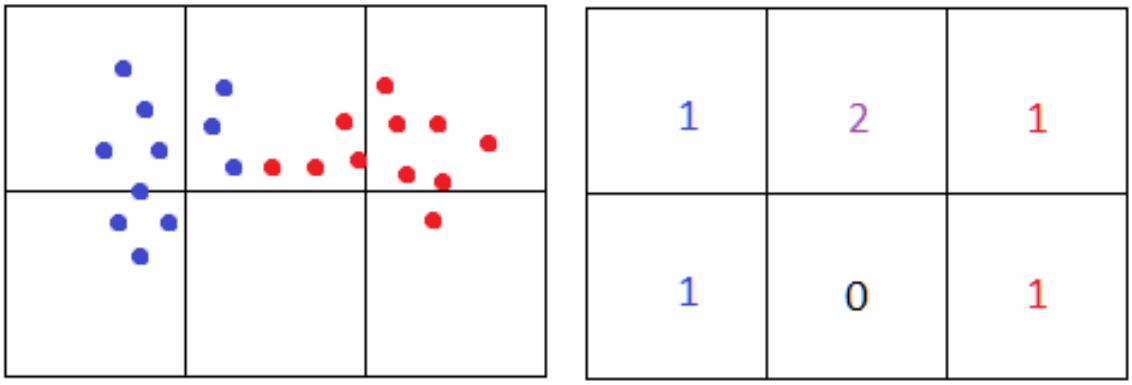


Figure 3.3: Assignment of Grid Occupancy values

4. ROBOT CONTROLLERS

4.1 Overview

This section describes the methods used to control the two types of robots in the scene: the HC10 manipulator, and the custom model mobile robot. First, the state machine for the HC10 is described in brief, followed by a description of the means to control it. Next, the mobile robot model is described in terms of its motion capabilities and control logic. Finally, remarks are made about how the two are brought together under a common server node that acts as a liaison between the robots and the Commander node.

4.2 HC10 Sander System

Software to control the HC10 Manipulator, currently having a sander attached to its end-effector, is available at the BARC lab[23]. The system makes use of ROS, to enable operation of the sander via both teleoperation (using a PS4 joystick controller) and a simple raster path generator that sends end-effector path to the manipulator to follow. The sanding operation is regulated using a state machine that divides the process into 5 states:

VALUE	STATE
S_0	Storage
S_1	Scanning
S_2	Move to part
S_3	Teleoperation
S_4	Scan to Path
S_5	Move to Storage

Table 4.1: Description of states in HC10's state machine

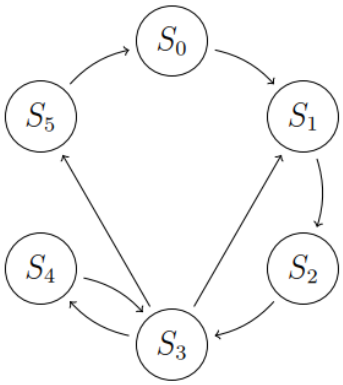


Figure 4.1: Succession of states in HC10's state machine

State 0 is the manipulator's 'storage' configuration, where the manipulator is safely tucked away onto its stand, to enable manual sanding, or loading/unloading of the workpiece with minimal obstruction. Each time a new part is to be sanded, a point cloud scan of the part must be made

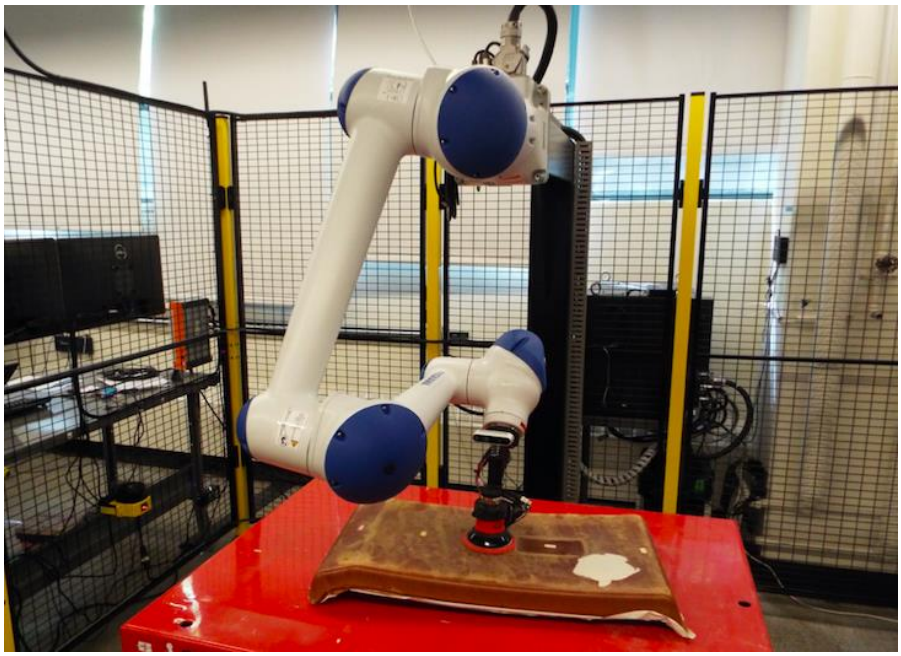


Figure 4.2: HC10 manipulator with sanding attachment at BARC Lab,

University of Washington

using an Intel RealSense camera mounted on it. Switching to State 1 triggers this process. Once the point cloud scan is complete, State 2 is reached, where the manipulator arm hovers at a position

over the arm. Switching to the next state, State 3, enables teleoperation control using the PS4 controller. By switching to the next state, State 4, the system automatically generates a path for the sander and performs the sanding operation. After sanding is complete, if a state change to State 5 is triggered, the manipulator moves to its storage configuration. The progression of states is shown in Figure 4.1.

4.3 Controllers for the Mobile robot

For this thesis, a model of an omnidirectional drive mobile robot was made. Figure 4.3 shows this model. Gazebo offers several plugins to control mobile robots, by means of a **libgazebo_ros_diff_drive.so** plugin to emulate a differential drive, **libgazebo_ros_skid_steer_drive.so** plugin to emulate a skid steer drive, and the more general, **gazebo_ros_control.so** plugin for a generic controller. Gazebo does not currently have plugins for omnidirectional drives, although people have got around this limitation by using multiple differential drives or skid-steer drives and casters. Thus, the mobile robot used here employs a similar method; it has 4 casters at the base, which can rotate within [-180^o, +180^o]. These casters are controlled using the gazebo_ros_control plugin. The wheels attached to these casters are then controlled using the Skid Steer drive plugin. For reference, the wheel diameters are 10cm, and the wheel base is 30cm. The four wheels are equally spaced about the centroid of the base of the robot. When building the robot model, the controller values for the 4 caster wheels were set to have proportional gain of 1000, integral gain of 0.01 and derivative gain of 1.0. These are default values for the controller, but they allowed the robot casters to respond quickly and turn to the desired angles, hence were not modified further. The skid steer drive plugin generates the rotational torque for the wheels when it receives Twist messages (rotational and translational velocity) over a /cmd_vel ROS topic.

4.3 Multi Robot Controls Server

As the HC10 and the mobile robot have different inputs, managing these inputs becomes a hassle when scaling the system to multiple robots, or even other types of robots. Furthermore, when carrying out path planning, it is cumbersome to initialize the controllers for each robot, develop

the control logic and generate low-level output that can directly be processed by these robots. To help solve this problem, a Multi Robot Controls Server node was created, that contains Python classes for each type of robot expected in the system. To add more robots, one simply needs to create a copy of the class and modify the body of some method calls. This server keeps track of the objects in the environment, and filters out mobile robots and manipulators. It initializes and starts ROS Action Servers for each robot, making 'goal' topics available for them, as an interface for the Commander.

For the HC10 Manipulator, messages sent to the 'goal' topic are either of type CALIBRATION or ON/OFF. If the message is of type CALIBRATION, the server cycles through states 1-4 of the HC10 state machine. When the state machine reaches state 4, the server also sends a message over a /motion_start_stop topic, signaling the manipulator to stop until it receives an 'ON' goal request. Thus, in state 4, this switch can be used to pause and resume machining operations on the HC10, which is particularly useful when pausing during debris detection in its proximity.

For the mobile robot, messages sent to the 'goal' topic are either of type CALIBRATION or POSE. Presently, no action is taken during the CALIBRATION phase, but when the POSE is provided, the mobile robot is expected to move towards it. This is done in 3 stages: first, the robot checks if it needs to rotate to a new orientation, and rotates if required. Next, it translates to the coordinates specified in the POSE. Finally, it does one final check for whether or not it is oriented correctly, and rotates itself if required. The values carried by the Twist messages, moderated by sliding mode control. At present, these values are as follows:

Translation		Rotation	
Velocity	Distance to target position	Velocity	Angle to target orientation
5 m/s	> 3 meters	2 rad/s	> 30^o
1 m/s	0.5 – 3 meters	1 rad/s	2 – 30^o
0.5 m/s	< 0.5 meters	0.5*(target_angle – current_angle) rad/s	< 2^o

Table 4.2: Twist values for Mobile robot using sliding mode control to control its velocity while following a goal pose

The Multi Robot Controls Server can thus be visualized by the infographic shown in Figure 4.3.

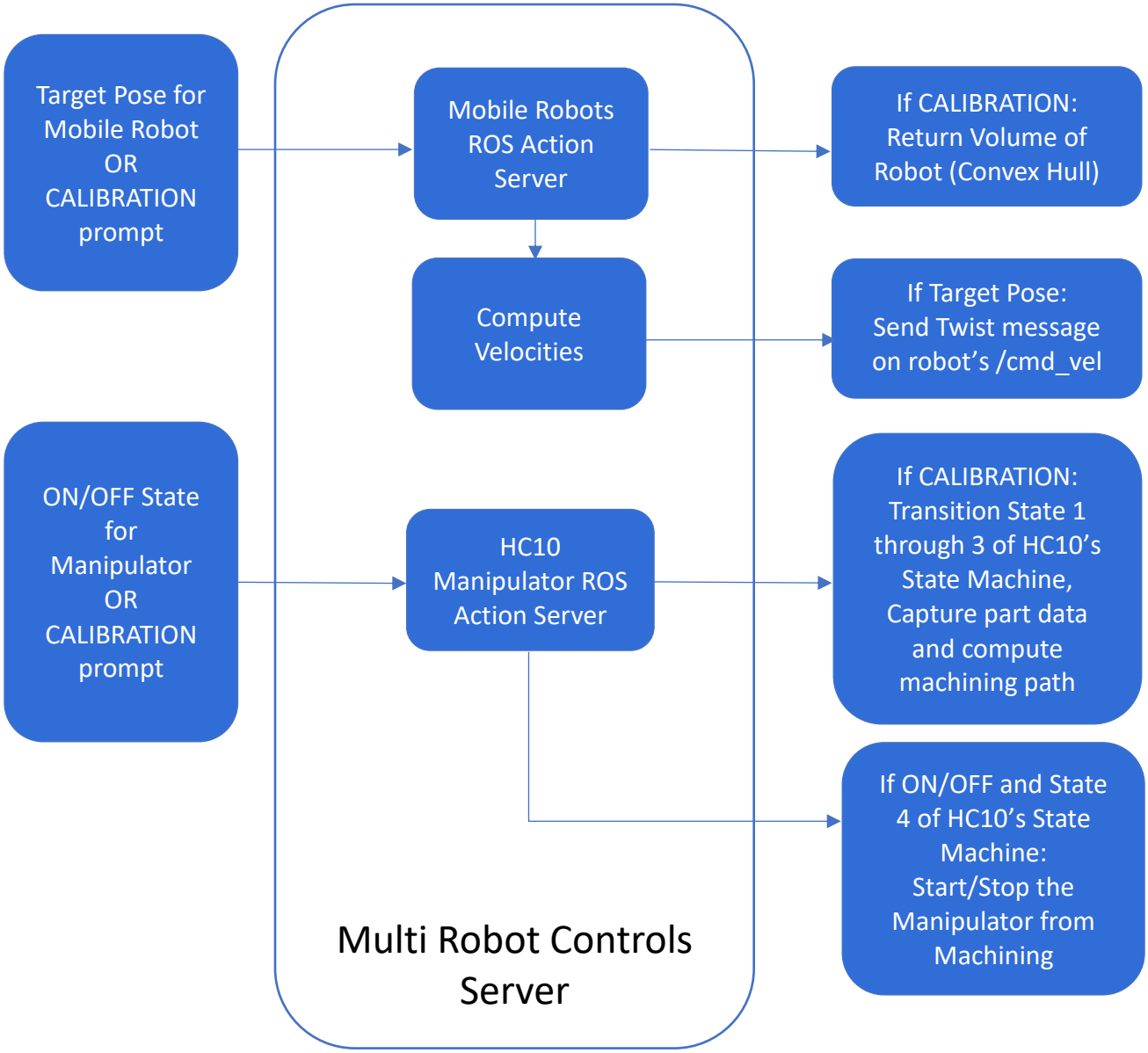


Figure 4.3: Multi Robot Controls Server overview

5. DATA STRUCTURES

5.1 Overview

This section describes the data structures used to store and handle various types of data transmitted through the ROS system, with a focus on how they help the Commander node's YARM planner execution. First, the Commander state machine node is introduced, followed by a brief description of its method calls and member objects. This is followed by a description of the TaskQueue and CostQueue data structures, which are the two primary means of organizing data available to the planner.

5.2 The Commander State Machine

The Commander is the central node of the system. Map metadata received from the database is unpacked and organized into 5 categories – Mobile Robot, Manipulator, Debris, Target and Obstacles. It provides the user a single interface to control all actions occurring in the environment automatically. The Commander is a state machine, currently consisting of 6 states, with potential to add more states as required. The states are outlined in Figure 5.1.

State transitioning is done by the user sending a ROS message, setting a Boolean value to either 'Previous State', 'Current State', 'Next State', or a numerical value to 'Go to State' variables in the message. Initially, the state machine is in State 0 (Idle), which occurs at start-up of the entire system. With any Boolean trigger, the state machine switches ON, transitioning to State 1. Nothing interesting happens up to here, but henceforth, the Boolean values in the ROS message will determine the state flow in the machine. By triggering 'Next State', the state machine switches to CALIBRATION mode. In this mode, a CALIBRATION goal is sent to the Multi Robot Controls Server for all robots that were unpacked from the map metadata. The state machine does not transition until calibration for all robots is complete. For a system consisting of the HC10, this means that the calibration state ensures that the workpiece is captured using the manipulator's on-board camera, and the manipulator is ready for machining. The state machine now offers the user to revert to the previous state, move to the next state, or stay at the current state, in which case, calibration occurs again. If 'Next State' is triggered, the state machine moves to an intermediate State 3, which was initially meant to serve as a menu for different modes of operation. One of the

modes of operation would be a ‘Bounded’ mode, in which the mobile robots were not allowed to interact with the manipulators. This would be state 4. However, for the purpose of this thesis, State 5 has been implemented, termed the ‘Boundless’ mode, in which mobile robots can move and

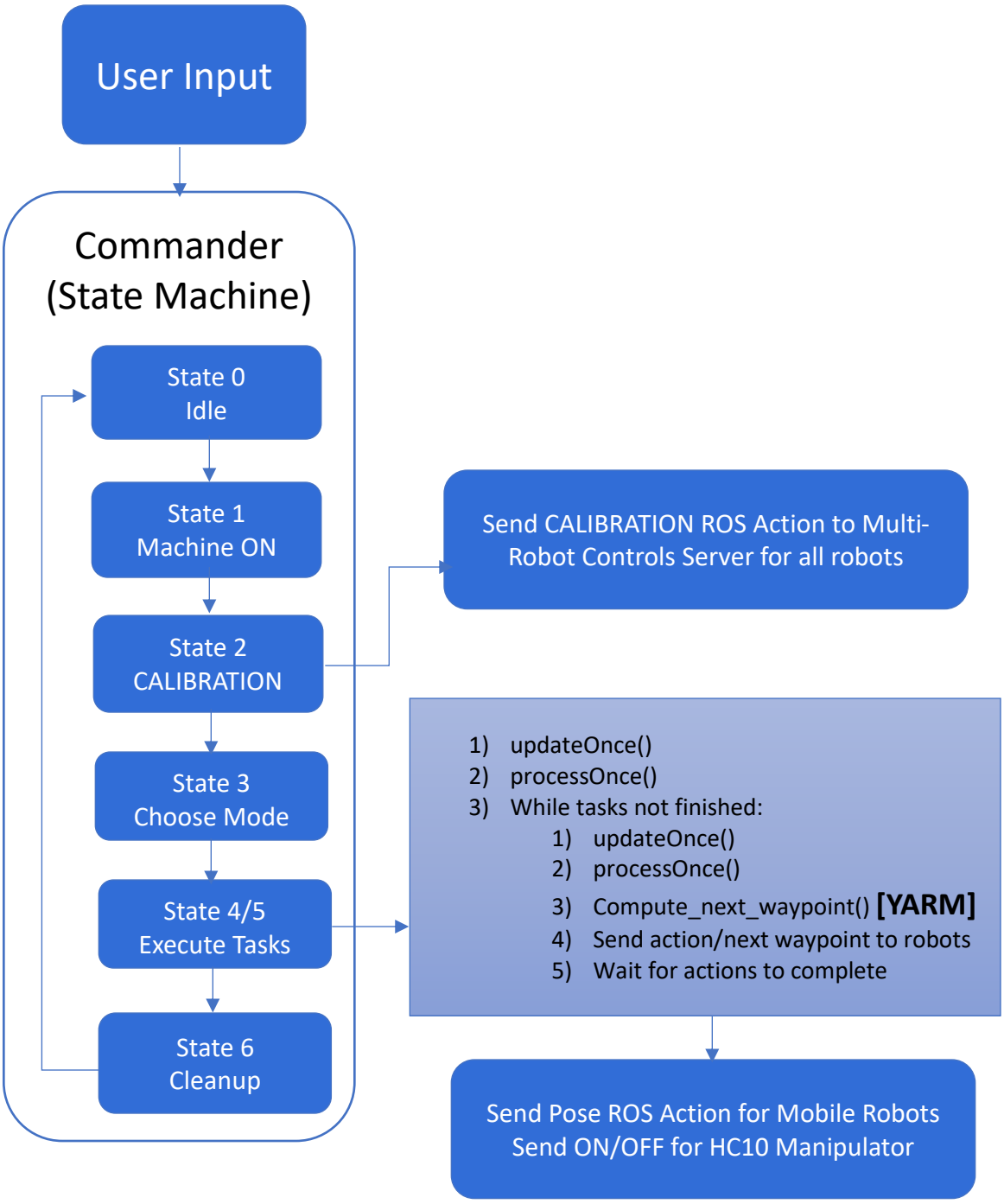


Figure 5.1: Commander State Machine overview

interact in the manipulator's surroundings. State 5 does a number of things, in sequence:

1) updateOnce()
2) processOnce()
3) while tasks are not finished
    a. updateOnce()
    b. processOnce()
    c. compute_next_waypoint (YARM)
    d. Send action/next waypoint to Multi Robot Controls Server
    e. Wait for actions to complete

The details of the updateOnce() and processOnce() function calls are explained in section 5.3. The details of compute_next_waypoint() will be covered in Chapter 6, along with an explanation of YARM.

Finally, once all tasks have been completed, the user has an option to transition the state machine to State 6, a 'clean up' mode. Currently, the state machine has been programmed to take no action in this state, but it acts like a placeholder for future work, such as possibly retracting robots and safely stowing them away to their charging ports, for example.

5.2 The TaskQueue and CostQueue

When State 4 or 5 is triggered, prior to YARM being executed, the mobile robots in the environment need to be assigned tasks. If there is debris in the environment, this is considered a priority, and the mobile robot closest to the debris is associated with it. Similar matching of robots with debris and targets in the environment is done using a TaskQueue data structure. This is a Python class consisting of a list of idle robots in the environment, a list of robot-target associated pairs, and two methods – updateOnce() and processOnce() that update information about these associations. At each call of updateOnce(), the robots, targets and debris information received from the metadata is used to update the positions of the robots, targets and debris in the list of associations. At each call of processOnce(), non-associated targets and debris are matched with idle robots, the association is added to the list of associated pairs, and the robot is removed from the list of idle robots. When a target has been reached, a pop() method of the TaskQueue class

deletes the robot-target association and moves the robot back to the list of idle robots. This acts like a simple priority-based scheduler, where the matching of robots with targets and debris could be done based on other criteria as well, based on the requirements of that industry. For the purposes of this thesis, the association is based on least distance to targets, which minimizes the changes of robots ever colliding into each other as they rarely cross paths.

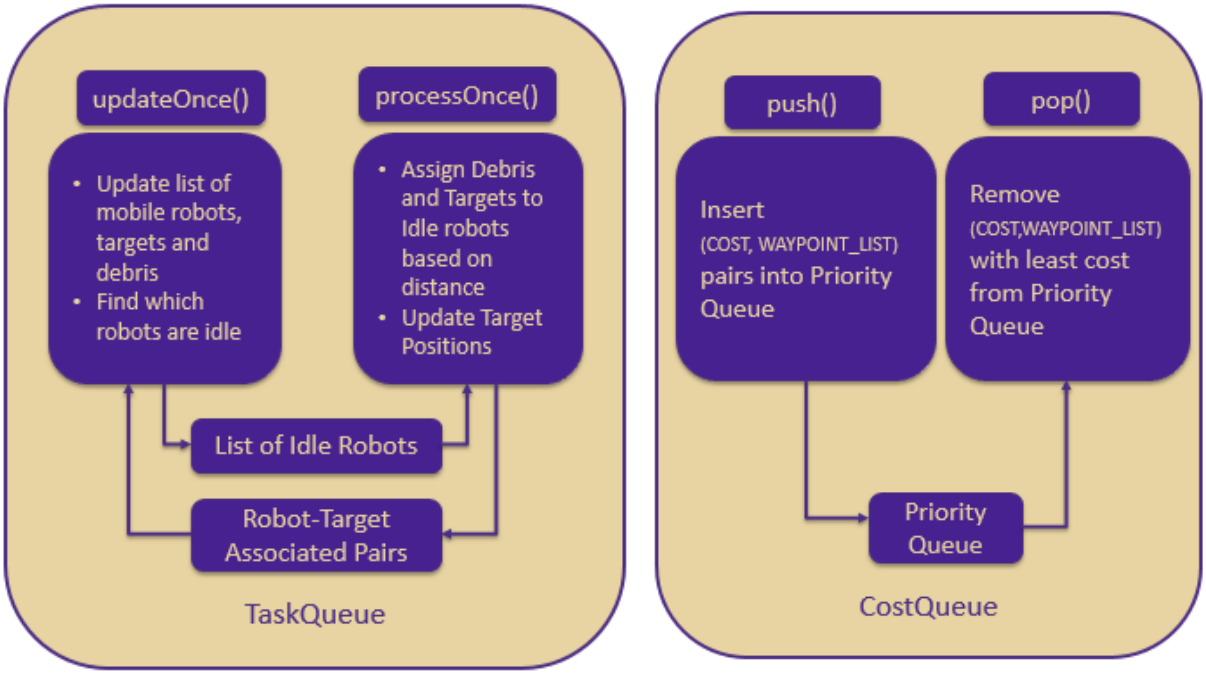


Figure 5.2: TaskQueue and CostQueue overview

The other data structure of interest is the CostQueue, which contains a priority queue and helps sort the input data by some metric. For the purposes of the YARM algorithm, the inputs to the CostQueue are pairs of (COST, WAYPOINT_LIST), where WAYPOINT_LIST is a list of dictionaries. Each dictionary contains information about the robot's pose, end-effector grid cell, list of grid cells the robot occupies, a robot 3D 'mask', and state of collision. The priority queue sorts the elements by COST, so that whenever a new element is inserted into this queue using the push() method, it is inserted at a position that matches its order of precedence, and whenever an element is removed from this queue using the pop() method, it is a (COST, WAYPOINT_LIST) pair with the least cost.

6. THE Y.A.R.M ALGORITHM

6.1 Overview

This section begins with a brief discussion of ideas from well-established path planning algorithms, followed by a methodology to expand the A* algorithm to a multi-agent case. This is followed by a description of the YARM algorithm and its components.

6.2 Common Path Planning Algorithms

The problem of path planning is not new, and several algorithms have been established, as well as incorporated into motion planning softwares such as MoveIt! The Navigation Stack in ROS makes use of A* to plan paths for mobile robots, using a combination of local and global costmaps. The global costmap computes paths to the goal, whereas the local costmap identifies if there is a need to redirect paths in a small radius surrounding the mobile robot, to deal with objects in the environment that may have moved since the first computation of the full path to the goal.

The important classes of path planning algorithms discussed here are Rapidly-exploring Random Trees (RRT), Probabilistic Roadmaps (PRM), A-Star, and their variants that are optimized for specific situations including dynamism in the environment. RRT builds a space-filling tree using samples randomly drawn from the search space, and is inherently biased to grow towards large unsearched areas of the problem. This generates a solution path, while avoiding obstacles. In some variants of this method, the trees are saved in memory so that in the event of dynamism, re-planning of paths can occur in local regions without having to search through the entire tree. RRT has been shown to be asymptotically optimal, which would mean multiple iterations over the tree structure are required to generate sufficiently optimal paths. A similar methodology of random sampling is employed in PRMs, where samples are drawn from the configuration space of the robot and a local planner is used to attempt to connect these configurations to other configurations. The start and goal configurations are known before-hand, are connected to the graph, and the path is obtained using a Dijkstra's shortest path query. Variants of the PRM algorithm, such as by Jaillet

et al[35] try to improve its performance in dynamic environments by pre-processing and storing valid paths with respect to static obstacles, and then use local planners to rapidly update the road map according to dynamic changes. Sampling-based planners, such as those developed by Fragkopoulos et.al [36] have also been researched for low to high DoF Manipulators, that have been shown to generate a good set of joint states over time, leading to collision-free paths, and strategies to optimize these paths are also being researched. The most important path planning algorithm, one that introduces the concept of dynamic programming for novices, is the A* algorithm. It makes use of a heuristic cost and priority queue to attempt to reduce the search space, and often generates an optimal path to the goal.

These algorithms cater to a wide range of motion-planning problems that require start-to-goal motion, but they differ greatly in terms of their applicability to this environmental setup. It must be emphasized, that the current choice of the ROS pipeline is intended to eliminate the need for multiple expensive sensors on robots, provide real-time 3D mapping and localization capabilities, as well as plan collision-free paths. In a sense, this can be considered a pipeline that simultaneously localizes, maps and plans paths for robots with multiple degrees of freedom, and attempts to perform motion planning by coupling the planners for mobile bases and their attached manipulators. Further, the occupancies of robots cannot always be approximated by bounding boxes as staying in a fixed bounded configuration at all times may affect their interactions with the environment. The mobile robot used for this thesis has a fixed arm attached to its base, and if a bounding-box approach was used for collision-checking, the robot would never be able to interact with the debris on the workpiece. A workaround for this would be to compute when the end-effector enters a specific region around the manipulator, and allow the end-effector to explore that region while performing collision checks for each explored point in that region. However, this works well when a map of the environment is known beforehand, so that the obstacles are known and plotted on the map. In the current environment, the map is not available before start of the simulation. For mobile manipulators especially, a live map is advantageous as it can help perform decision-making in real time instead of waiting for the map to update object occupancies based on sensor readings from individual robots, and subsequent localization by iterative filtering.

6.3 Multi-Agent A*

During the initial phase of this research, attempts were made to develop a means to perform path planning for multiple agents using A* search. A* was chosen because of known optimality in paths provided there is a solution. Like all path-planning algorithms, this algorithm was built to compute on a simplified version of the environment, consisting of a 2D grid containing occupancy values denoted by either 0 or 1. The challenge with adapting A* for multiple agents, was to decide how to deal with collisions or conflicts during the search. One way to do this, was to update a list with the next states of each robot in turn, so that a robot that plans its step last, won't move to any of the states in that list, nor to states that it has already visited before. If a robot runs out of choices, it stays at its current position and waits for the next update on the map. The set of waypoints computed up to the current state are added, along with total cost (path length + heuristic) to a priority queue called the Agenda. This process can be summarized by the flowchart in Figure 6.1. It is to be noted that this is not the only means of doing so, but is one approach that worked well and provided a similar idea of prioritizing robots, which is used to develop the YARM algorithm.

6.4 The YARM Algorithm

At its core, the YARM algorithm is a multi-agent graph search algorithm with reduced search space. The search space does not remain constant throughout the system; it updates every time the robots reach their next state. Dynamic re-planning makes use of the saved planned path from the previous search space, and compares it with the newly computed path from the new search space,

so that the robots get to choose their next states by comparing the two paths and opting for the state with lower cost and no collisions.

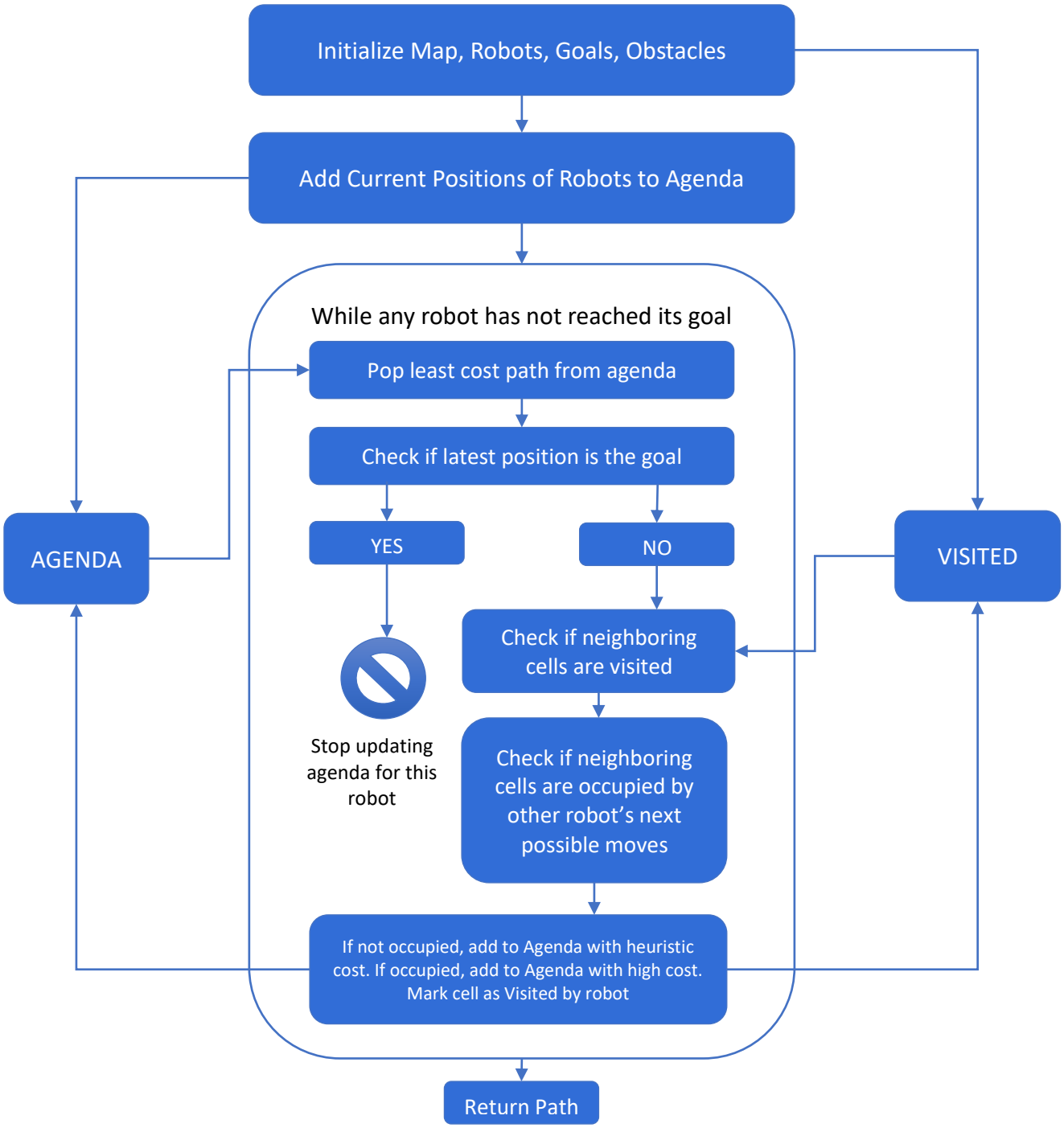


Figure 6.1: A methodology to adapt A* for the multi-robot case

When compute_next_waypoints() is called, the Commander has already loaded all robots, obstacles, debris and targets into memory, and the TaskQueue has assigned debris/targets to idle robots. If there are more robots than debris or targets, some mobile robots will not perform any tasks and remain idle. The case where there are more targets or debris than mobile robots, has not been implemented yet and is left as a future exercise. With this available data, compute_next_waypoints() executes in 3 stages:

1. Initialization Stage: This is where all the robot-target and robot-debris pairs data is saved into a temporary dictionary, their old set of waypoints, if any, are extracted from their previous CostQueues, and their CostQueue is set to NULL. Next, particles are 'thrown' onto the map, which, in this case, means placed into a set. The mobile robots' end-effector's occupied grid cells, and the target/goal grid cells are also added to this set of particles. A graph is generated from the particles, using Delaunay Triangulation[37]. The robot's current poses are appended to an empty list and this list is added to the Agenda.
2. Generating Waypoints: The grid cells that the robots occupy, are vacated from a copy of the 'live' 3D map, so that there exists a robot-less map. This robot-less map serves the same purpose as the list of current states in the Multi Agent A* in Section 6.3. Through a loop, the optimum list of path waypoints is popped from the Agenda, the latest pose of each robot is taken, and the neighbouring particles corresponding to that pose are searched for collisions. The particles are individually appended to a separate copy of the current list of states, and added to the Agenda with the latest computed cost. This loop runs until every robot's end-effector has reached its goal grid, or the Agenda becomes empty.
3. Dynamic Re-planning: Once the optimum list of waypoints to the goal is received for the current call of compute_next_waypoints, the waypoints are compared against a previous set of waypoints, if available. The comparison criteria are the path lengths of each, and whether or not the robot expects to collide at the immediate next step. The better path is selected and saved as the optimum list of waypoints. The next pose for the robots is generated from this list and sent to the robots.

The full algorithm is described below:

Initialization Stage

1) robot_less_map = timestampmap = None, particles = [], Robot_dict = {}, currentmap, [$robot_i$, $target_i$], storage = {}, recent = {}, define MAX_ITER, Graph = None, define configurations
2) robot_less_map = removeRobotGridsFromMap(currentmap, $robot_i$) for all i
3) if type($target_i$) == Debris for any $target_j$,
    a. robot_less_map = removeRobotGridsFromMap(currentmap, $target_j$)
4) timestampmap = copy(robotlessmap)
5) Add GridCell($robot_i$), GridCell($target_i$) to particles for all i
6) Sample n points from the 3D grid and add to particles
7) Graph = DelaunayTriangulation(particles)
8) Save $robot_i$, $target_i$ information in robot_dict in this format:
    a. Visited = []
    b. Agenda = CostQueue() (empty costqueue)
    c. CurrentWaypoints = []
    d. Optimum = None
    e. initPose, initGrids, goalGrid <- obtain from map metadata

f. Iterations = 0, Status = False, Target = $target_i$,, NextPose = None

g. Storage[$robot_i$] = None, recent[$robot_i$] = None

h. COST = distance_heuristic(GridCell($robot_i$), goalGrid)

i. ROBOT_S_CURRENT_WAYPOINT = [(GridCell($robot_i$), initPose, initGrids, Collisions=False)]

j. Agenda.push((COST, ROBOT_S_CURRENT_WAYPOINT))

Generating Waypoints

1) While True

a. For all $robot_i$

i. If isEmpty(Agenda), set Status to True and continue

ii. Else Optimum = pop(Agenda), storage [$robot_i$] = Optimum

1. If Optimum[1][-1] == goalGrid, set Status to True

2. Iterations += 1

iii. AddRobotGridsToMap(timestampmap, $robot_i$, Optimum[1][-1])

b. If Status is True for all $robot_i$, break

c. If Iterations > MAX_ITER for any $robot_i$, break

d. For all $robot_i$, if Status is False,

i. Temp = removeRobotGridsFromMap(timestampmap)

ii. Neighbors = get_neighbors(Graph)

iii. For p in Neighbours

1. If p in Visited, pass
2. Else
    a. Heuristic_cost = distance_heuristic(p, goalGrid)
    b. Translation = vectorDifference(p, GridCell($robot_i$))
    c. For config in random(configurations), do getTransformations()
        i. Obtain newGrids, Cost (for rotation, translation), newPose, newMask, vlessMask, vlessGrids
        ii. Colliding_cost = collision_heuristic(newmask, timestampmap)
        iii. Find configuration with least total cost = Colliding_cost + rotation cost + translation cost
    d. Olist = copy(Optimum[1])
    e. Append p to Olist, Visited
    f. COST = distance_heuristic(p, goalGrid)
    g. Agenda.push((COST, Olist))

Dynamic Re-planning

1) For each $robot_i$
    a. If recent[$robot_i$] is None
        i. Recent[$robot_i$] = Optimum[1]
        ii. If Optimum[1][1][Collisions] is False
            1. NextPose = Optimum[1][1][initPose]
        iii. Else
            1. NextPose = initPose
    b. Else
        i. Old_pathsum = sum(Recent[$robot_i$])
        ii. New_pathsum = sum(Optimum[1])
        iii. If Old_pathsum < New_pathsum
            1. If Old_pathsum[Collisions] is False
                a. NextPose = Old_pathsum[1]
        iv. Else
            1. If New_pathsum[Collisions] is False
                a. NextPose = New_pathsum[1]
            2. Else
                a. NextPose = initPose

```
3. Recent[robot_i] = Optimum[1] if New_pathsum <
   Old_pathsum
```

6.5 Methods used in the Algorithm

This section briefly explains what some of the method calls in the algorithm do, for better clarity

1) *removeRobotGridsFromMap()* – marks the robot's currently occupied grid cells as vacant
2) *addRobotGridsToMap()* – marks the input grids as occupied on the given map
3) *DelaunayTriangulation()* – converts the input set of points into a graph using Delaunay Triangulation. The output Graph can be queried using an input point, and the neighbours of that point are returned as a list
4) *isEmpty()* checks if an Agenda is empty
5) *random(configurations)* samples configurations from the configuration space of the robot
6) *getTransformations()* transforms the robot's current set of occupied grid cells to new positions that would be occupied if the robot were to move to a new configuration. The method also returns the newly transformed grid cell indices, poses and masks.
7) *newGrids, newMask* are the robot's grids and masks that mark each cell on the map that the robot would occupy during transition of its configurations, as occupied. *vlessGrids, vlessMask* only mark those cells on the map that the robot would occupy after it has reached its final configuration.
8) *Collision_heuristic()* returns a high value if grid cells from two or more different objects in the environment coincide (collide)
9) *Distance_heursitic()* returns the Euclidean distance between two points

7. EXPERIMENTS, OBSERVATIONS AND RESULTS

7.1 Overview

This section of the thesis describes the experiments performed using the YARM algorithm. It provides visualizations and observations of the paths generated, whether a solution was reached, and whether collisions occurred, from 3 different types of experiments. The effect of dynamic re-planning on the quality of generated path is discussed results from these experiments. A single robot-target case is also investigated to understand the effect of number of particles on the computation time per step as well as total computation time.

7.2 Experiment 1: One Mobile Robot, One Debris, One Manipulator (1M1Ma Case)

In this experiment, a single mobile robot was allocated the task of moving towards a single debris in the machining path of the manipulator. Experiments were conducted by placing the robot at different points in the environment. The observation of interest here was that the manipulator successfully stops in proximity of the debris, and the mobile robot reaches the debris while avoiding collisions with the manipulator arm.

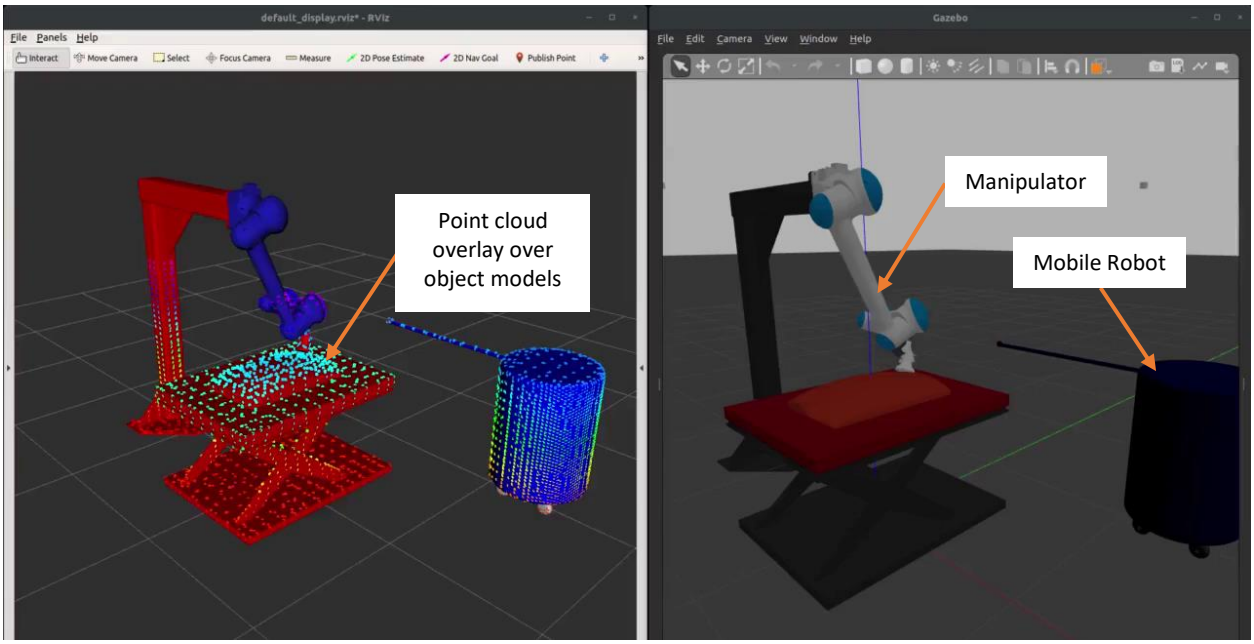
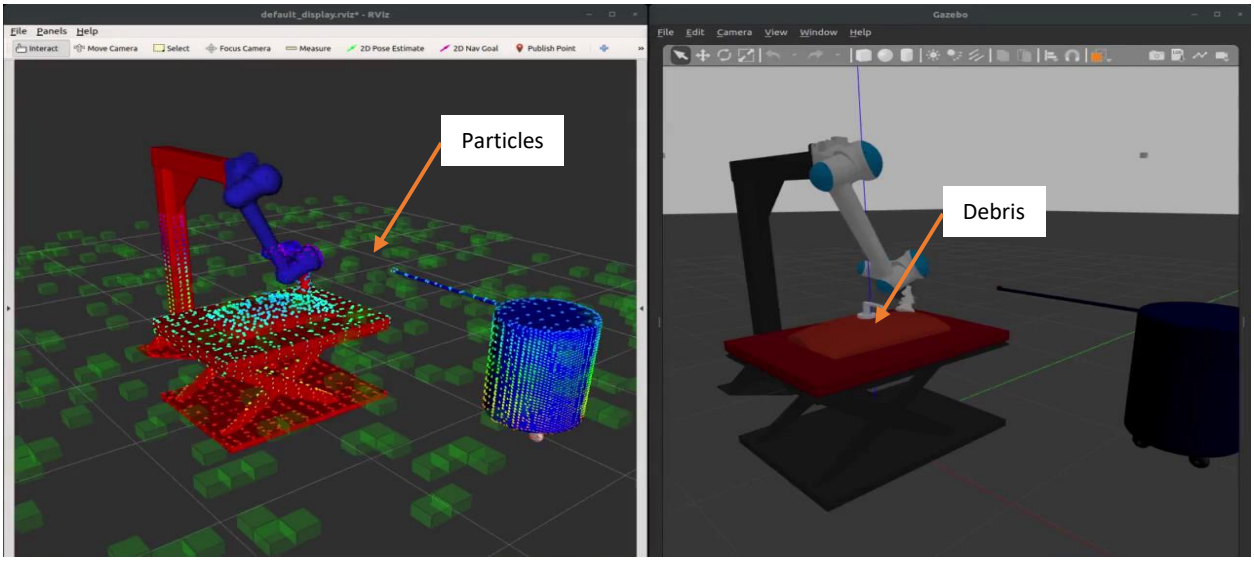
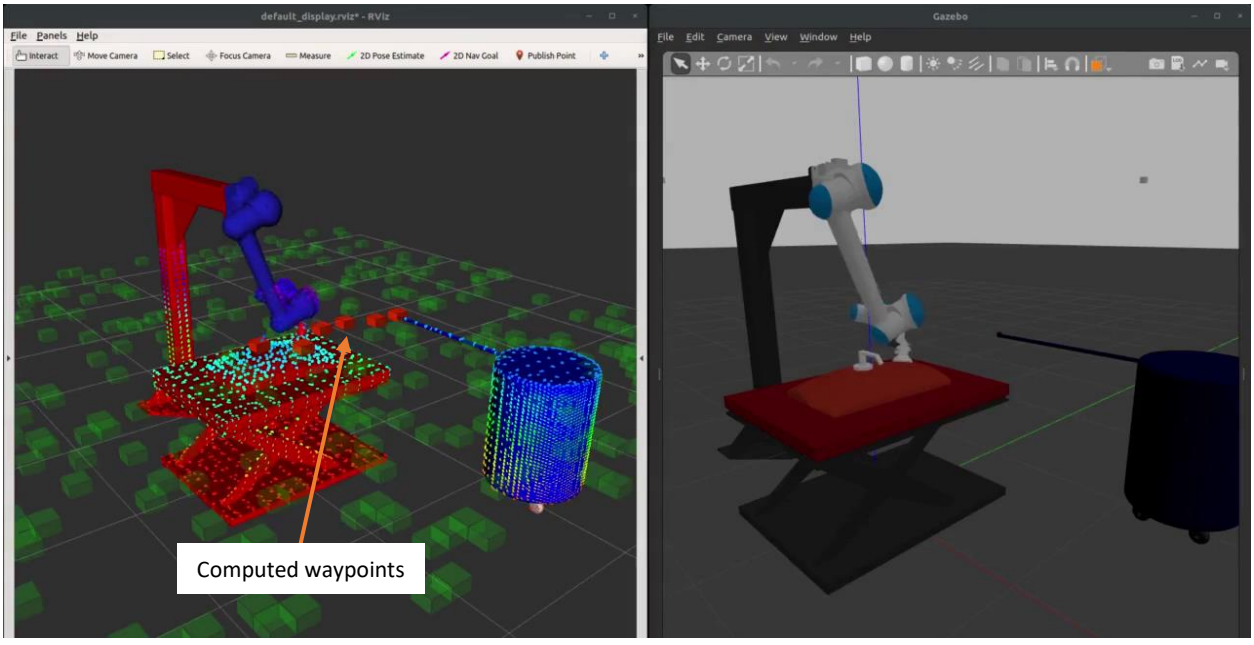


Figure 7.1: A side-by-side view of RViz and Gazebo environment for the 1M1Ma case

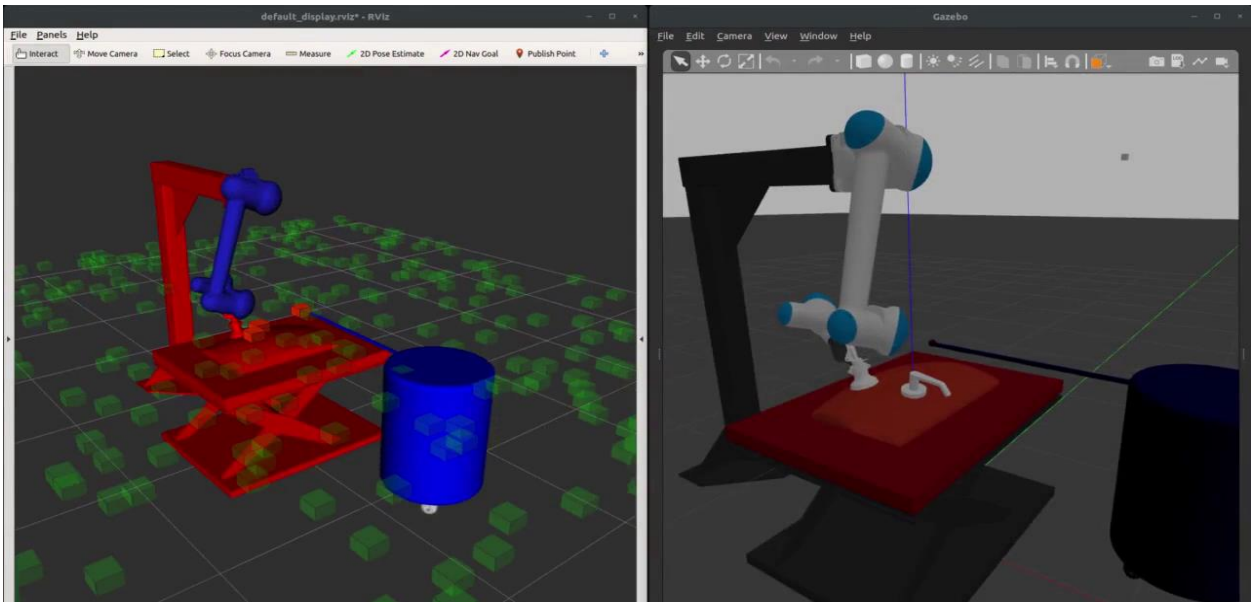
Figure 7.1 shows a side-by-side view of the environment as displayed in RViz (left) and Gazebo (right) with the lone mobile robot and manipulator, before the start of computation and spawning of debris. Figure 7.2 shows the progression of the experiment, by capturing points of interest.



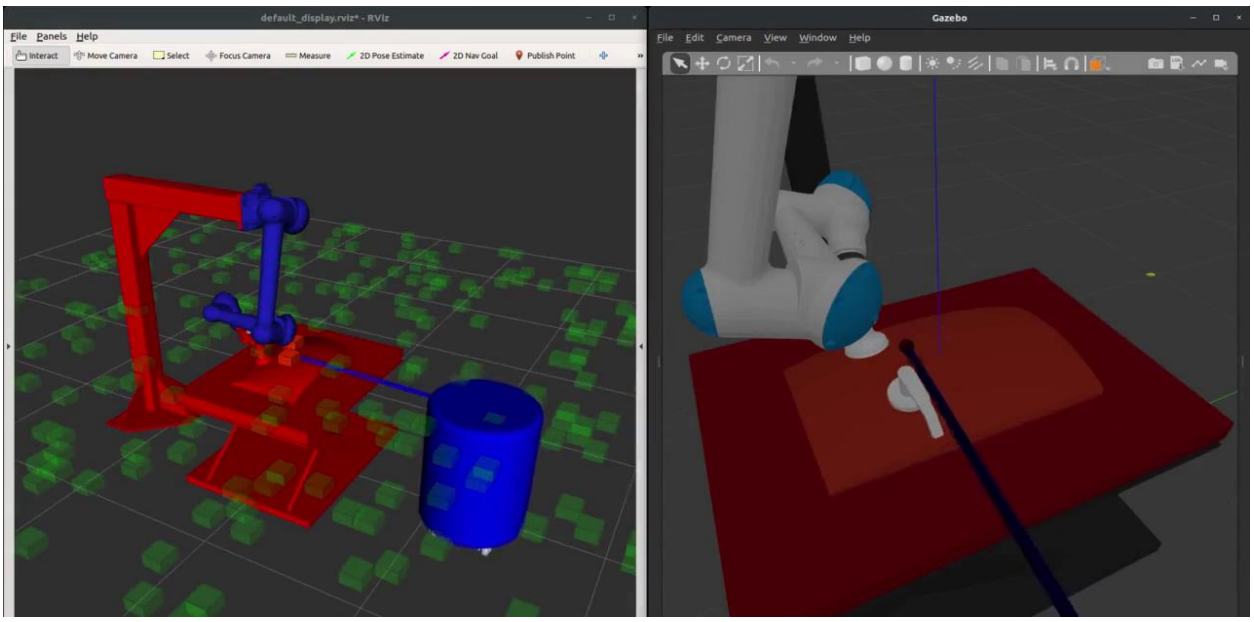
(a)



(b)



(c)



(d)

Figure 7.2: Frames of interest in Experiment 1

(a) Debris is spawned on the workpiece

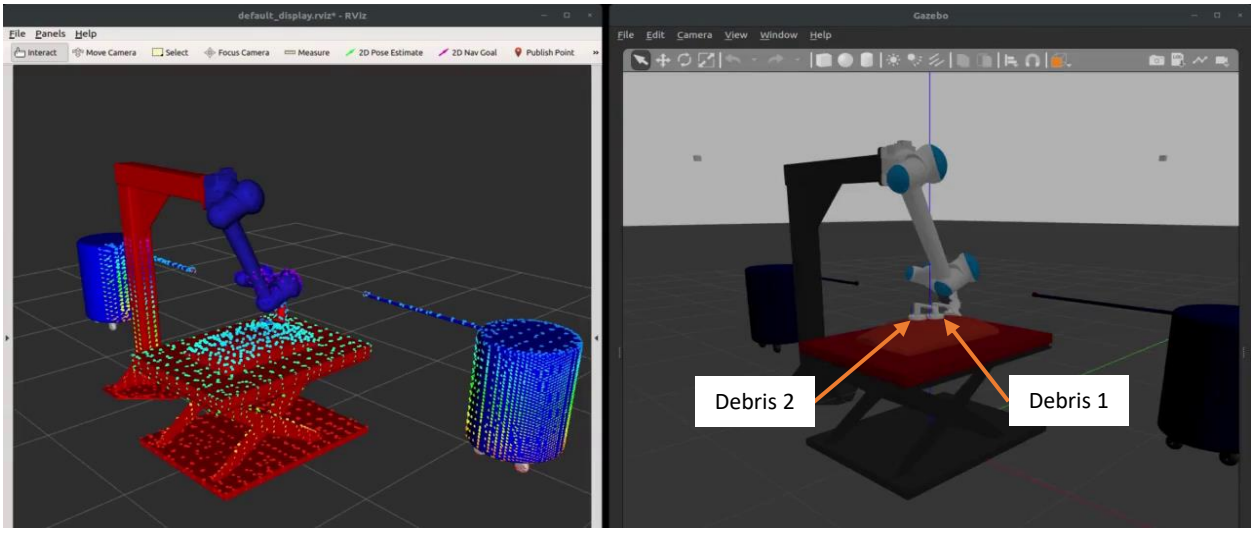
(b) Waypoints are computed by YARM; Manipulator starts machining

(c) Manipulator detects debris in its proximity and pauses

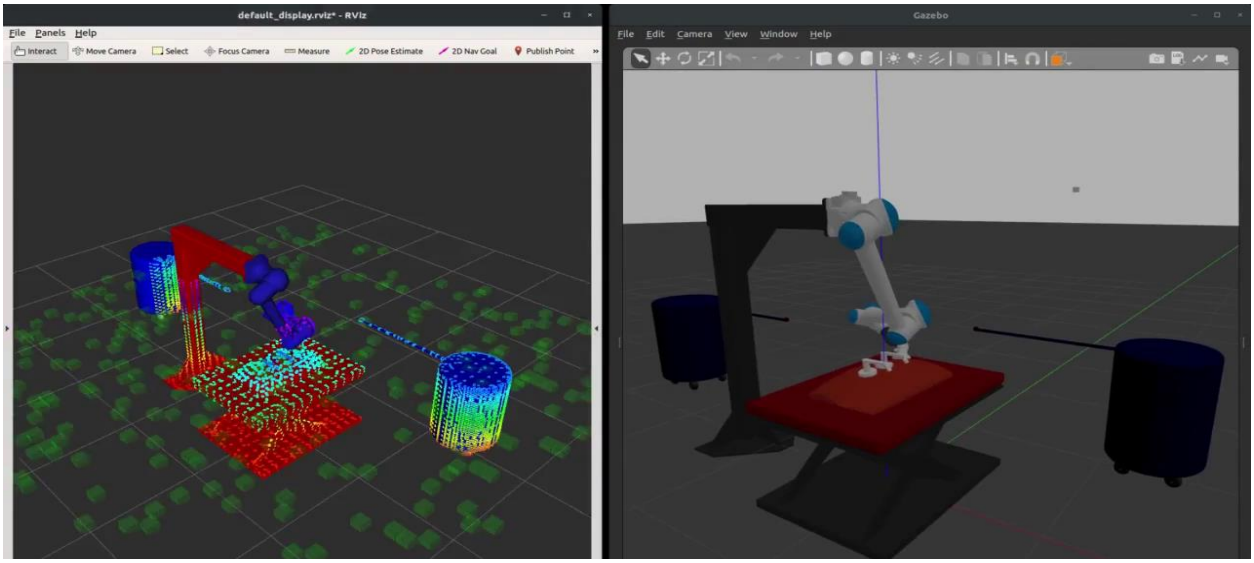
(d) Mobile robot has reached the debris

7.3 Experiment 2: Two Mobile Robots, Two Debris, One Manipulator (2M1Ma Case)

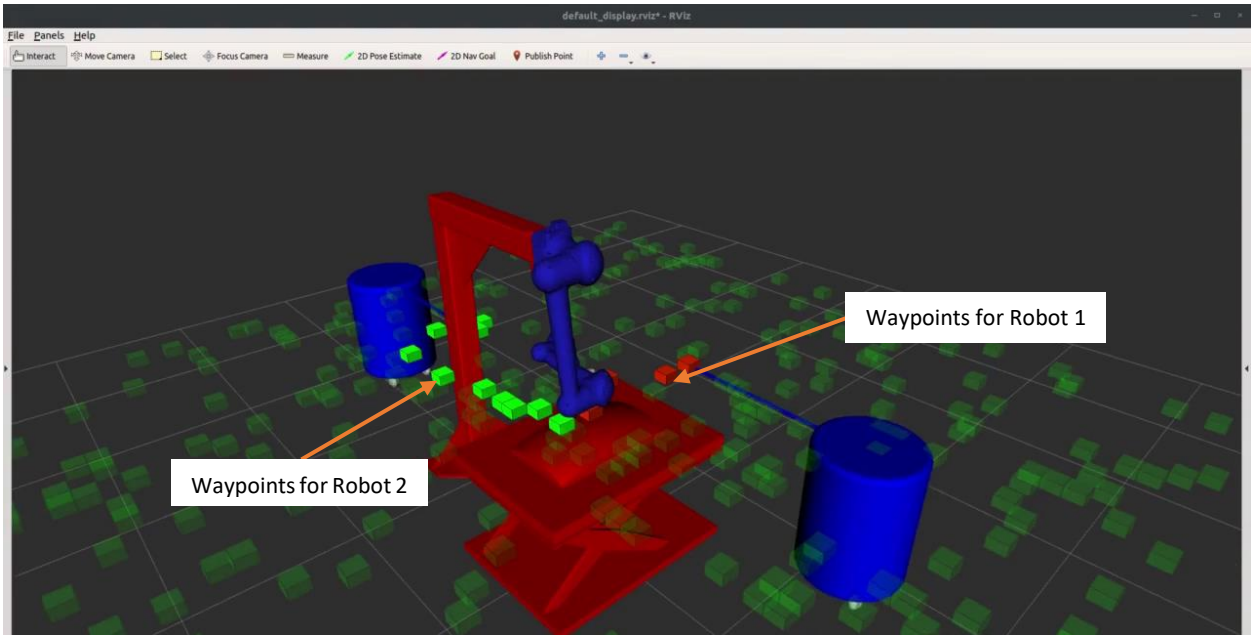
Experiment 2 scales up Experiment 1 by introducing an additional mobile robot and debris. Experiments were conducted by placing the mobile robots at different positions. This experiment has the additional complexity that sometimes the debris closest to a robot won't be reachable by that robot, as the robot is constrained to not rotate. The TaskQueue does not take this into account, and therefore, does not evaluate reachability of the goal. So the experiments were conducted using reachable goals for each robot. Figure 7.3 shows the progression of the experiment with key points of interest.



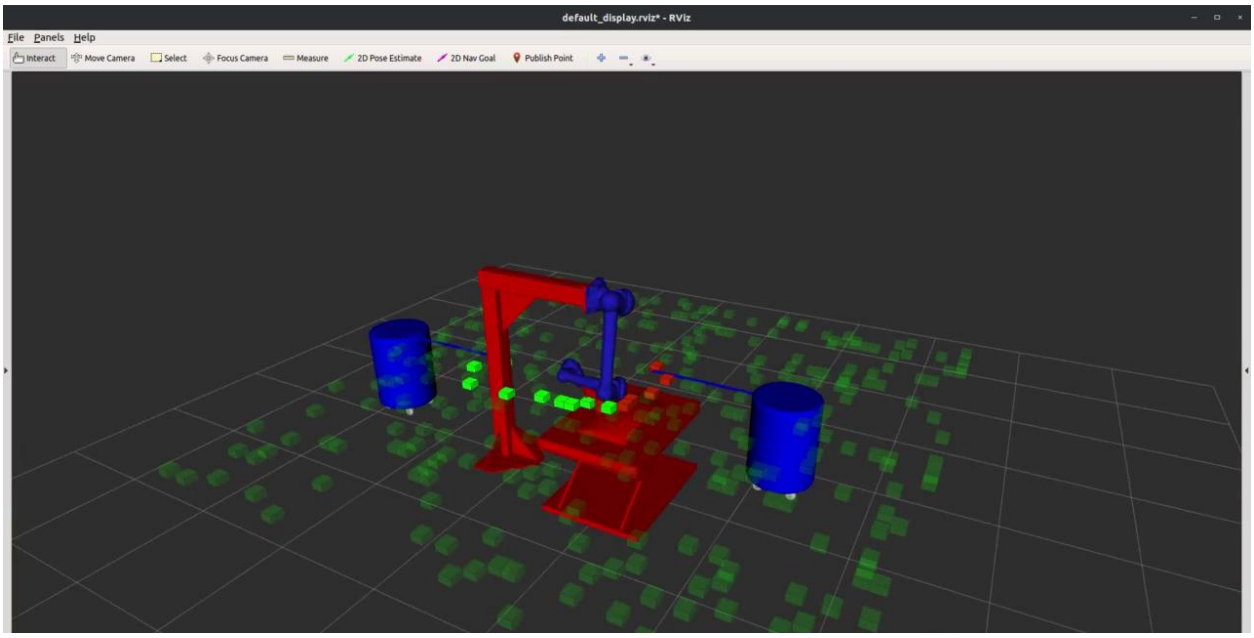
(a)



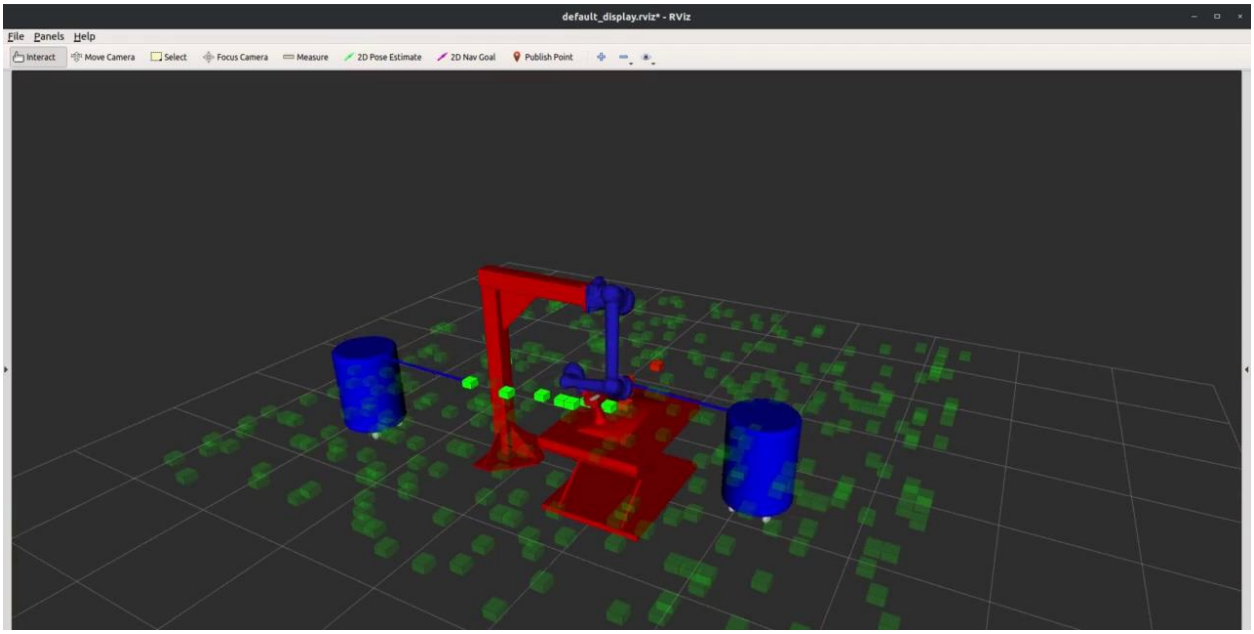
(b)


default_display.rviz* - RViz
File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Nav Goal Publish Point
Waypoints for Robot 1
Waypoints for Robot 2

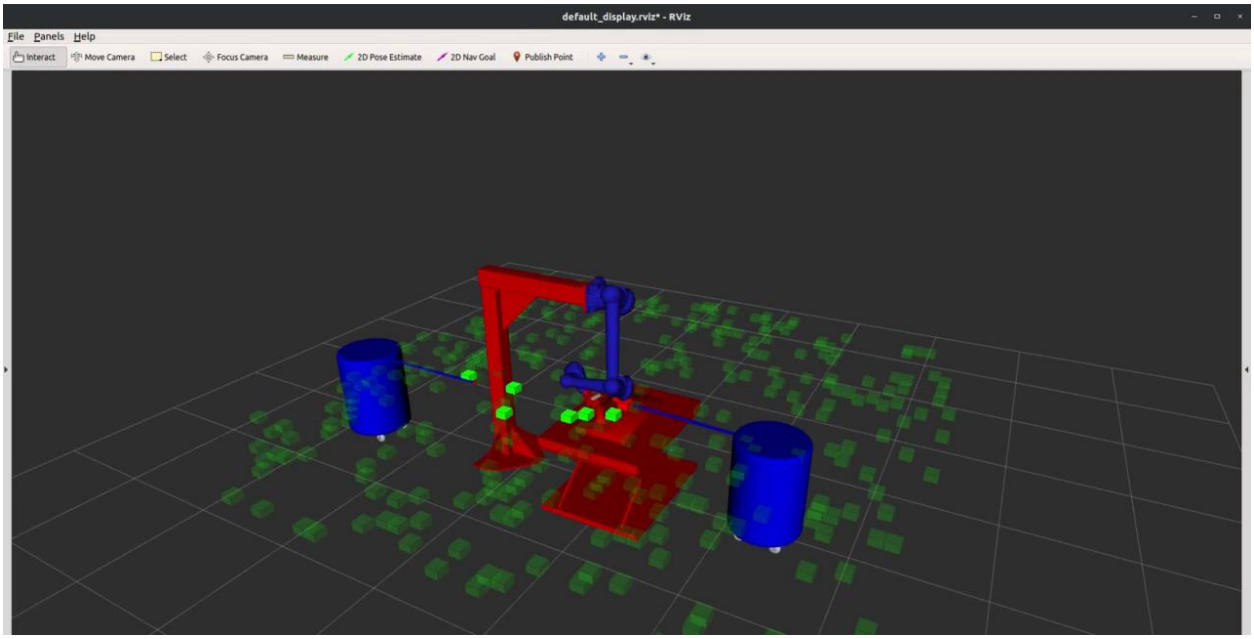
(c)


default_display.rviz* - RViz
File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Nav Goal Publish Point

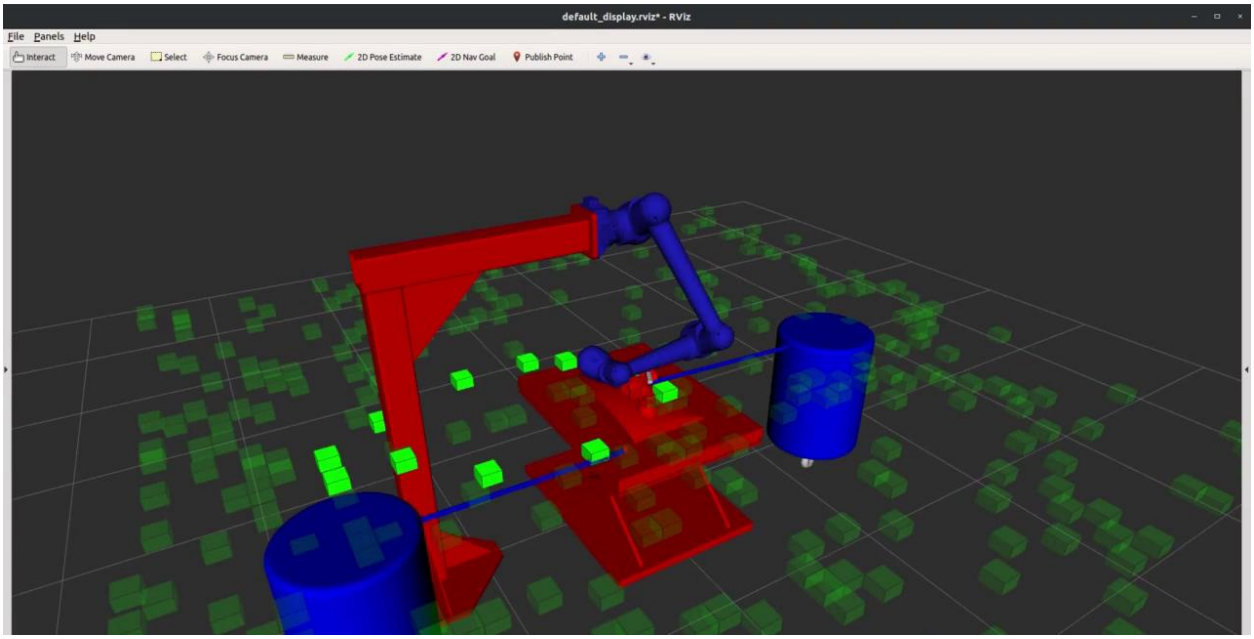
(d)


default_display.rviz* - RViz
File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Nav Goal Publish Point

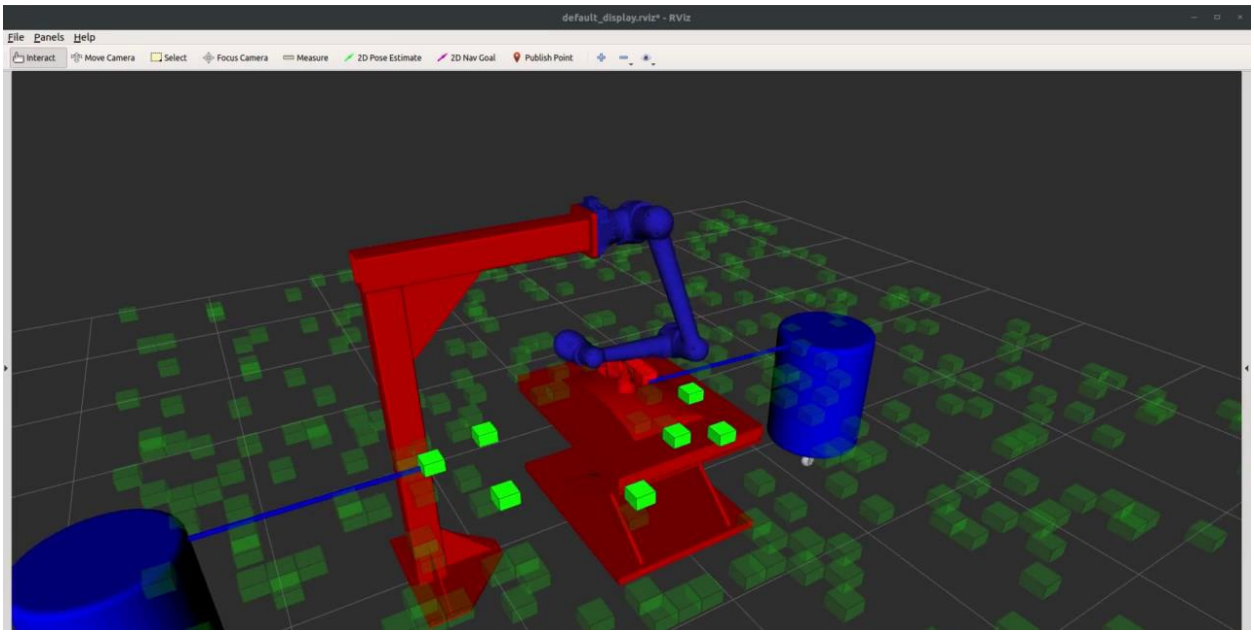
(e)


default_display.rviz* - RViz
File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Nav Goal Publish Point

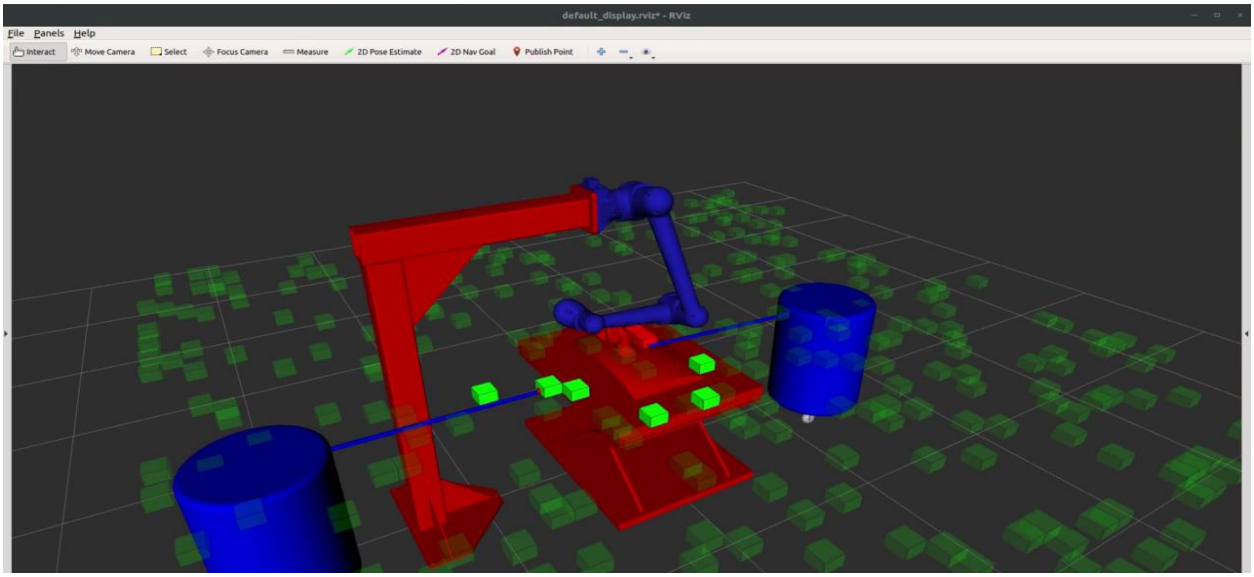
(f)


default_display.rviz* - RViz
File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Nav Goal Publish Point

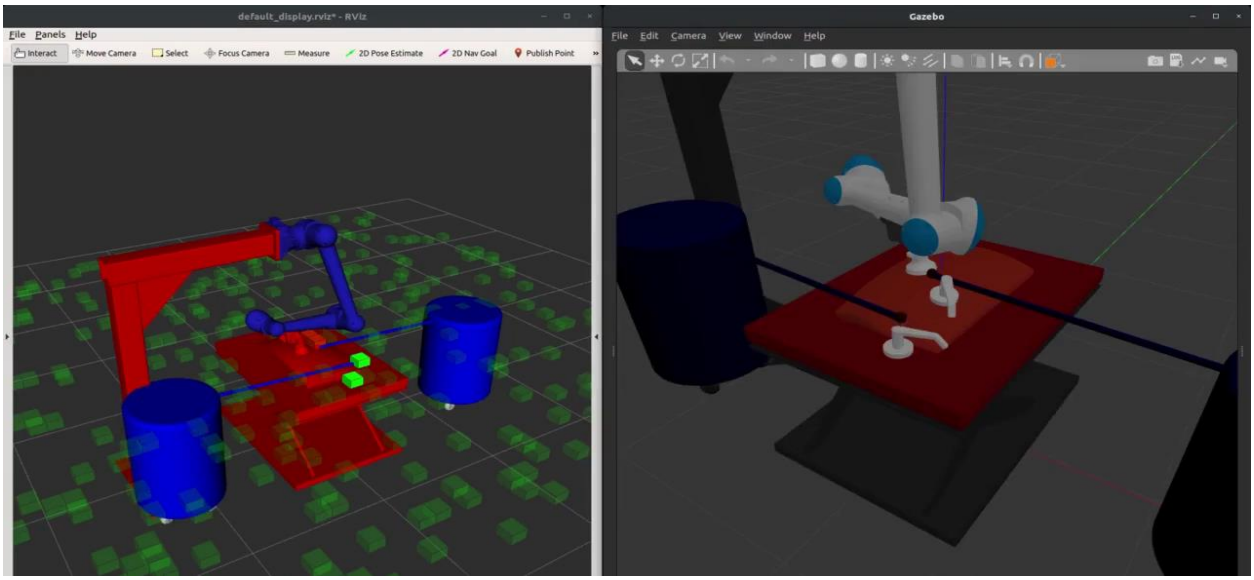
(g)


default_display.rviz* - RViz
File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Nav Goal Publish Point

(h)



(i)



(j)

Figure 7.3: Frames of interest in Experiment 2

(a) 2 Debris are spawned, (b) Particles are thrown on the map, (c) Waypoints computed for mobile robots, (d) Manipulator stops near one of the debris, (e) Debris has slipped, goal pose updated, (f) manipulator moves again and obstructs path of mobile robot, (g) Mobile robot re-computes path to goal, (h) Debris slips again, manipulator moves and stops near another debris, (i) Mobile robot re-computes path to goal, (j) All mobile robots have reached their assigned debris goals

7.4 Experiment 3: Three Mobile Robots (3M Case)

In this experiment, the manipulator is safely tucked away, so that the table, workpiece and manipulator with its vertical stand behave as obstacles. The controllers for the manipulator are not loaded, so this emulates a case where mobile robots have to only navigate obstacles in their paths. Since a multi-agent case is being dealt with, three mobile robots were initialized at different positions in the environment, and paths to their goals computed by YARM. The TaskQueue matches robots with their closest target goals, so in most cases, this minimizes the chances of paths crossing each other. Figure 7.4 shows important results from this experiment.

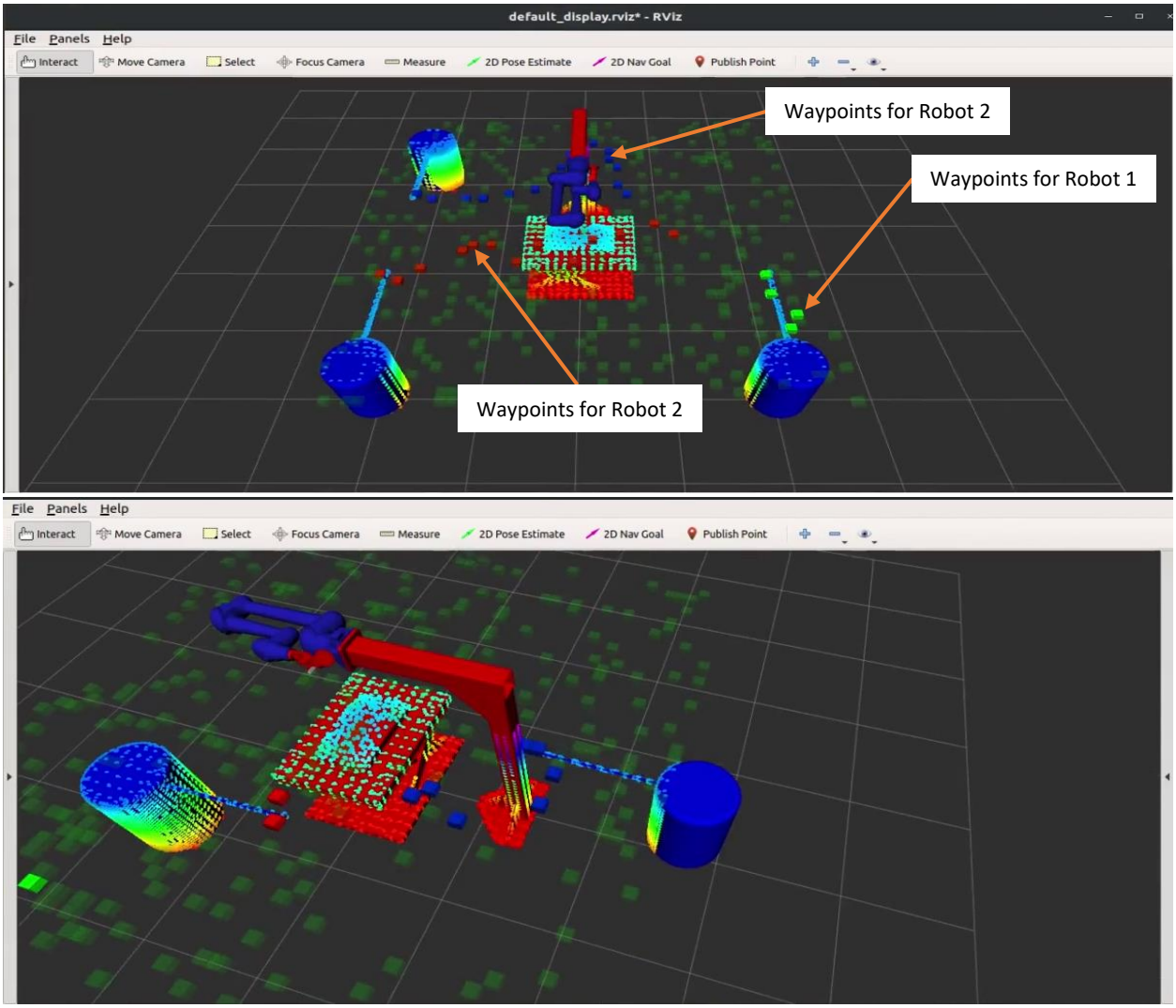
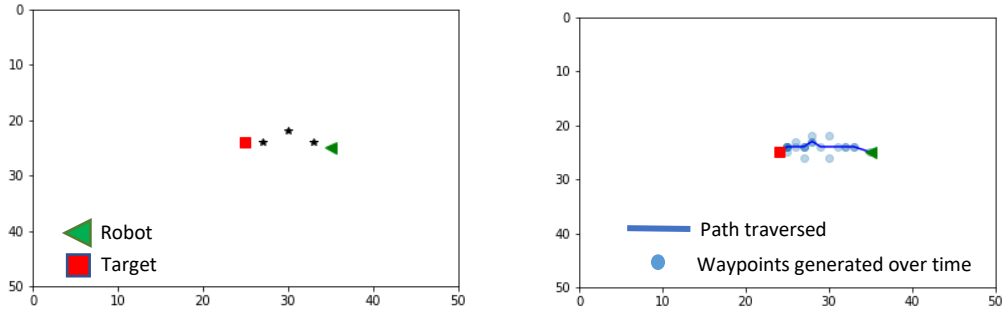


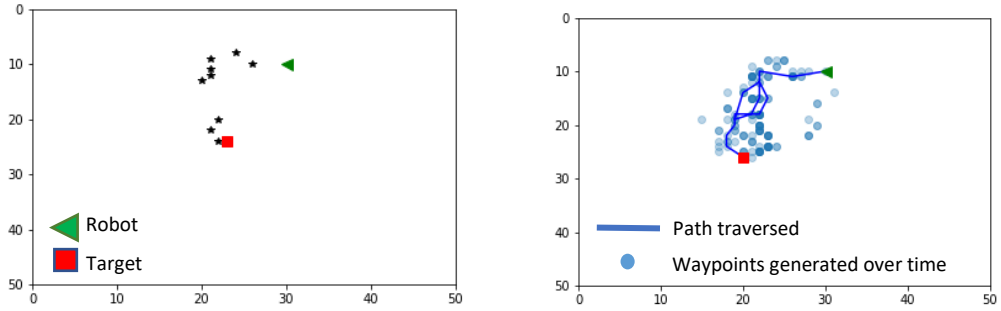
Figure 7.4: (Top) Waypoints computed for the 3 mobile robots. The waypoints ensure the robots don't collide with each other and with the environment obstacles. (Bottom) Robot 3 successfully avoids collisions with the vertical stand before reaching its goal

7.5 Results and Discussion

From the experiments conducted in Sections 7.2 – 7.4, it can be seen that the mobile robots successfully avoid collisions with each other and the manipulator, and YARM computes paths to the debris for each of them. Having such paths computed and executed is a successful solution to the path planning problem posed in the experiment. It is interesting to note how the paths get dynamically refined at each step that the robots move. The data from the 2M1Ma and 3M cases are plotted in Figure 7.5 and 7.6 respectively. This shows a 'heat map' of waypoints that were computed over time, as well as the path that the robots actually executed over time. The motion of the manipulator is not plotted on these graphs as it is irrelevant to this discussion.

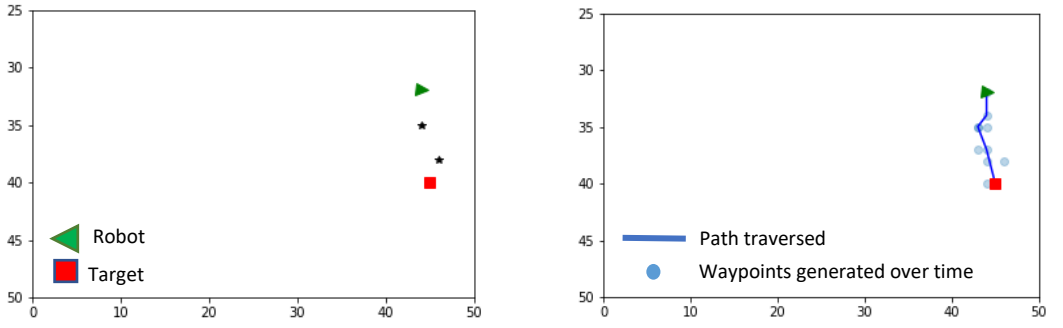


(a)

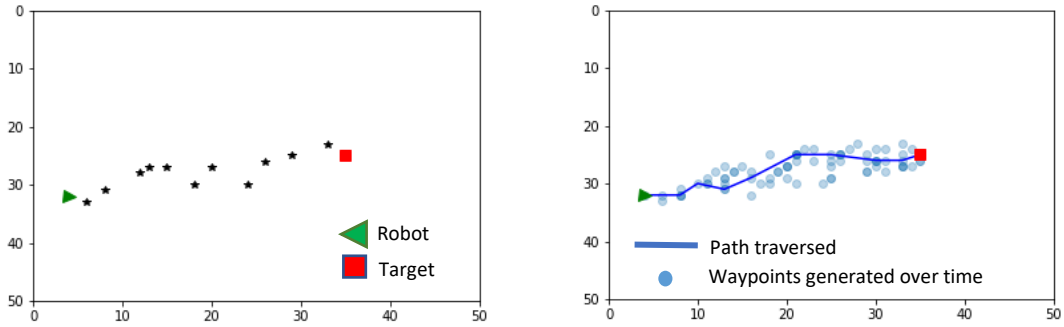


(b)

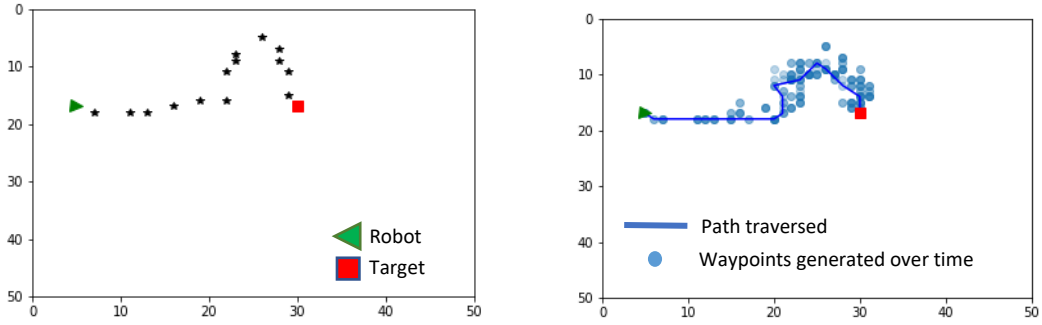
Figure 7.5 : Waypoints and path computed for the 2M1Ma case – (a) Initial set of waypoints (Left) and actual path followed (Right) for Robot 1, (b) Initial set of waypoints (Left) and actual path followed (Right) for Robot 2. X-Axis and Y-Axis represent grid indices in space. The loop shows the robot's re-planning after the debris slipped from its initial position.



(a)



(b)



(c)

Figure 7.6 : Waypoints and path computed for the 3M case – (a) Initial set of waypoints (Left) and actual path followed (Right) for Robot 1, (b) Initial set of waypoints (Left) and actual path followed (Right) for Robot 2, (b) Initial waypoints (Left) and actual path followed (Right) for Robot 3. X-Axis and Y-Axis represent grid indices in space.

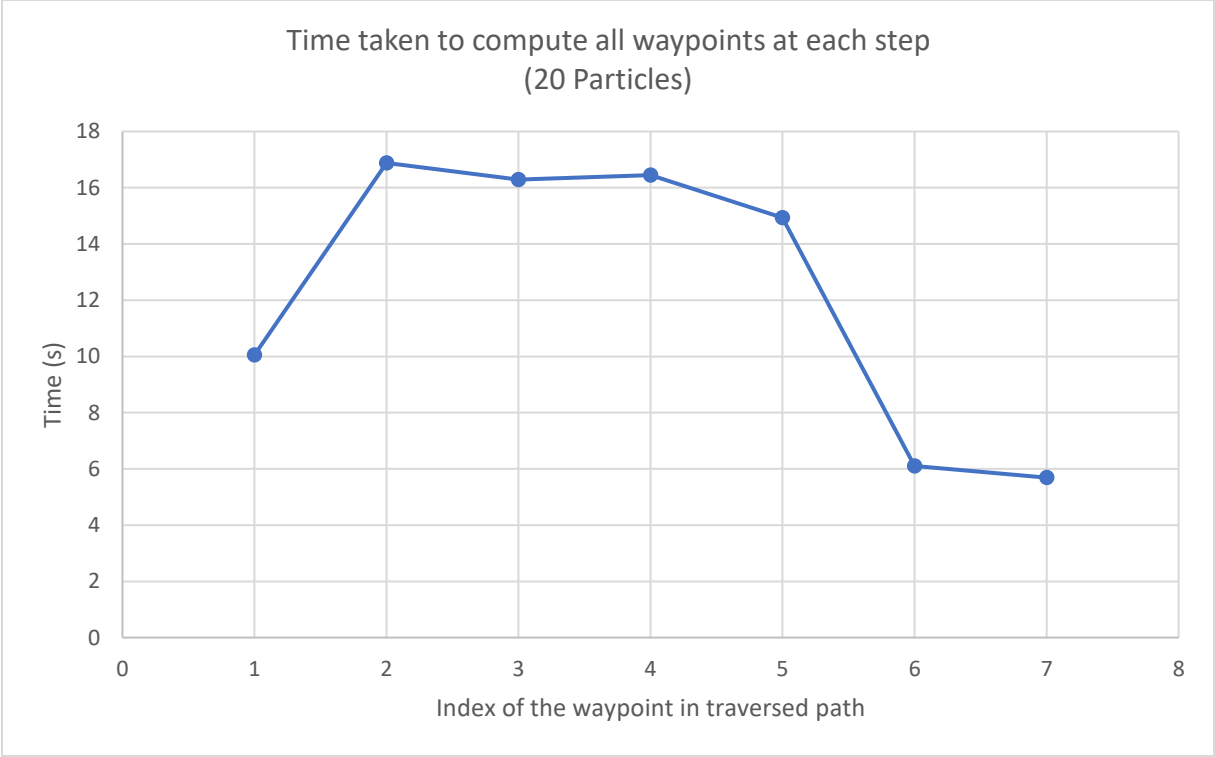
An observation about the environment can be made, that it is sparsely cluttered. Workspaces are usually kept organized, so that information about stationary obstacles in the environment is always available at calibration time. This is an important assumption at the start of execution of the Commander. Thus, YARM is useful for such sparse environments where it is beneficial to skip checking for collisions at several points on the map.

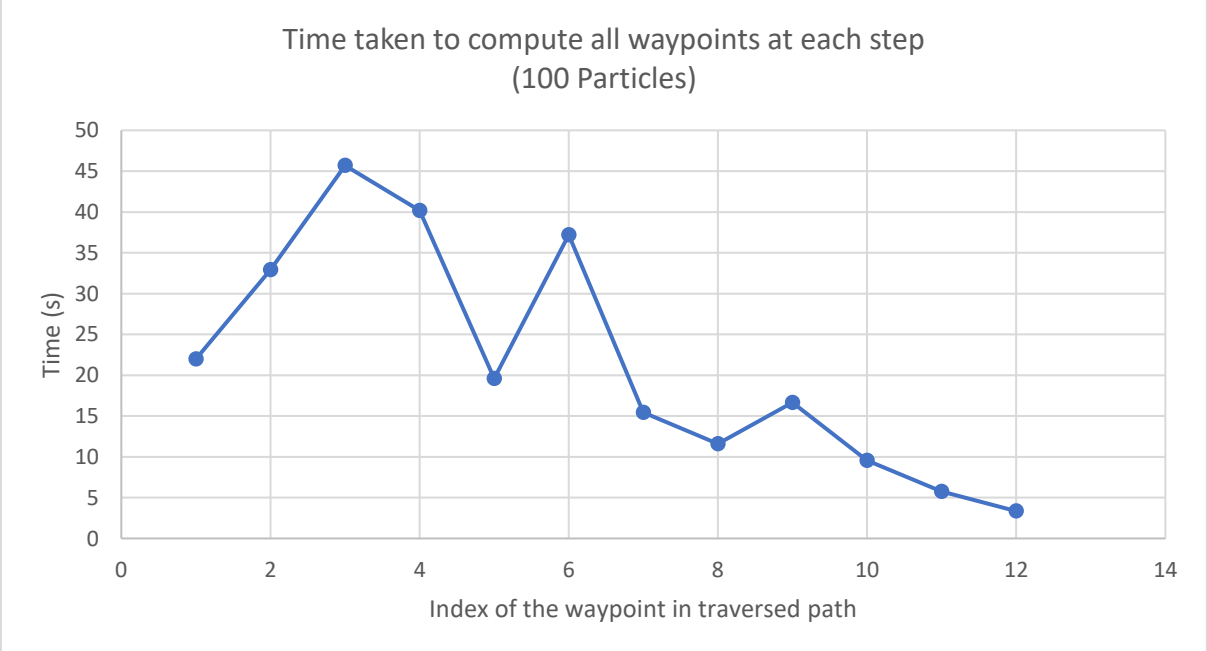
For dynamic environments, YARM works well, as, by using the real-time map it can automatically update the states of objects in the environment and use that for planning. Prediction algorithms may be used for highly dynamic objects with uncertainty in motion, so that those states can also be updated on the map. This would be useful for making YARM work with humans.

From the experiments, it can be seen that YARM works well with multiple robots by using the provided ROS pipeline. As particles are distributed randomly in space, it may not plan optimal paths for very short distances, but as can be observed, the paths tend to move towards optimum over time. This optimality can be improved by modifying the path-comparison algorithm and using better alternatives for memoization during dynamic re-planning. While YARM does not immediately provide the most optimal solution during the first iteration, and if it does, it doesn't provide a metric to check optimality, it can be observed from the results that the paths generated are sufficiently good to drive robots to debris. As debris removal is more time-critical than requiring optimality in paths for sparse environments, YARM shows excellent potential for this use case.

YARM is tunable in terms of the number of particles that are thrown into the environment. For a single mobile robot having to avoid a single obstacle to reach its target (no debris involved), the time taken and number of iterations to move the robot to its goal, using YARM, was observed by varying the number of particles. This would give an insight to what the time complexity looks like with dynamic re-planning. For reference, the initial position of the robot body was at (2m, 2m, 0m) and goal pose was at (-2m, -2m, 1.0m). Figure 7.7 shows the plots for these observations.

It can be observed that as the number of particles increases, it takes more time to compute waypoints to the goal. The saturation case occurs when all particles generated correspond to every


Time taken to compute all waypoints at each step
(20 Particles)
Time (s)
0
2
4
6
8
10
12
14
16
18
0
1
2
3
4
5
6
7
8
Index of the waypoint in traversed path


Time taken to compute all waypoints at each step
(100 Particles)
Time (s)
0
5
10
15
20
25
30
35
40
45
50
0
2
4
6
8
10
12
14
Index of the waypoint in traversed path

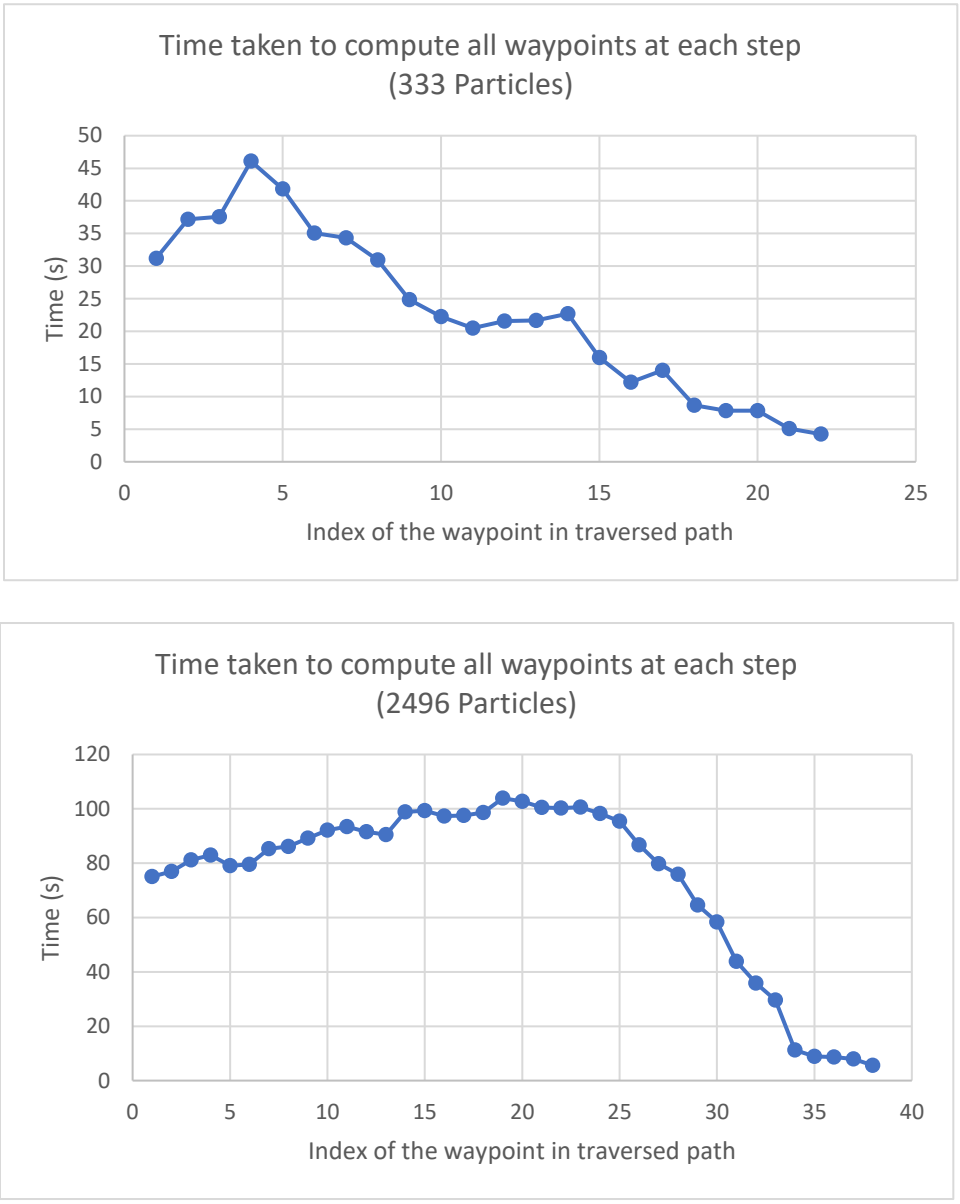


Figure 7.7: Effect of number of particles on computation time. The plots display the index number of the waypoint during path traversal by the mobile robot, versus the time taken to compute the remaining waypoints to the goal, when the robot is at that waypoint.

point in the environment, in which case, the YARM resembles performing an A* search. Thus, it can be observed that by using fewer particles, YARM performs better than an A* search in terms of search space. It is also observed that as the robot gets closer to its goal, the time taken to compute the remaining waypoints to the goal, drastically reduces. This could be attributed to the smaller

search space (of particles) as the robot nears its goal, and also the robot having computed an optimal set of waypoints at some positions and not diverting from that set as it moves on.

For another set of experiments, the manipulator was switched on, and no debris was spawned in its path. However, the mobile robot was provided a target (away from the manipulator) that it had to reach without colliding with the manipulator as it machined the part.

Dynamic Replanning?	**Number of Particles**	**Number of Waypoints**	**Collisions with Stationary Obstacles?**	**Collisions with Dynamic Objects?**	**Robot Stuck?**	**Solution? (Path from start to goal computed successfully)**
No	20	5-8	Yes	Yes	Yes	Sometimes
	100	10-14	Sometimes	Sometimes	Sometimes	Sometimes
	333	20-25	Sometimes	Sometimes	Sometimes	Sometimes
	2496	35-40	No	Sometimes	No	Yes
Yes	20	5-8	No	No	Sometimes	Yes
	100	10-14	No	No	Sometimes	Yes
	333	**20-25**	**No**	**No**	**No**	**Yes**
	2496	35-40	No	No	No	Yes

Table 7.1: Observations about planned paths using YARM with and without dynamic re-planning, for a single mobile robot, moving towards a non-debris target while avoiding the HC10.

Observations were made regarding whether or not the robot collided with static or dynamic obstacles, or whether the robot got stuck in a loop. These experiments were conducted by two methods: with and without dynamic re-planning. Table 7.1 summarizes these results.

It can be concluded that dynamic re-planning is essential for YARM in dynamic environments, else the robot tends to collide with dynamic obstacles if it keeps following a pre-computed path.

8. FUTURE WORK

8.1 Limitations

The field of Path Planning has well-established solutions for mobile robots for a lot of different applications, however, most of these applications deal with low degree-of-freedom mobile robots, and therefore, these robots are used only as movers. When we consider the general problem of path planning with mobile manipulators, especially ones with large configuration spaces, it is difficult to arrive at an optimal solution. Sometimes multiple optima may exist, so decision-making would have to be involved to choose one of the solutions. The problem statement, of reaching a debris, can be compared to the general problem of reaching an object to grasp or manipulate for given tasks, and this general problem is also limited by the similar factors – multiple path solutions, more constrained environments, constrained motions, dynamism and unpredictability in the environment, at the very least. In all cases, a combination of algorithms would have to be suitably modified and used in tandem to solve a smaller sub-set of problems that are highly specific, but still have limited or conflicting performance when faced with another sub-set of problems.

With regards to the experiments performed using YARM, there were limitations, as will be addressed here. At present, the simulation entirely runs on a laptop with an Intel i7 2nd Generation CPU with 8GB RAM, which isn't sufficient compute power to run all pipelines in real time. Gazebo Simulator is resource-intensive, and runs all simulations on the CPU. At present it does not have GPU optimization. Thus, Gazebo Simulator was running at a Real Time Factor of 0.08-0.1, which means that the simulations ran at a tenth of the speed that they should in real time. With Gazebo claiming a bulk of the compute power, this leaves very little resources for point cloud processing, the remainder of the ROS messaging system, and, in turn, the YARM algorithm computations. This means that running Gazebo on another compute resource would vastly improve compute times observed. The second resource-intensive task is the state machine of the HC10 manipulator, which should ideally be run on its own compute resource. In a factory environment, the compute infrastructure is expected to be better organized, branching out compute power to optimize performance. Thus, the compute times for YARM with dynamic re-planning are reasonable enough for the resource it was tested on.

With regards to the algorithm itself, it presently uses a simplified mobile robot restricted to rotate. When YARM uses rotations of the mobile robot to determine good configurations of the robot, this increases the number of collision-checks it does at each waypoint, which increases computation time. For a better compute resource, this would not pose a problem, but for the current platform, the compute times increase by a factor of approximately 1.2 to 2 per robot in the simulation. The selection of particles also affects the performance of the generated path; fewer particles would generate paths faster, but do not necessarily guarantee a solution, nor high optimality of the path itself. However, this is due to the type of decision-making currently implemented during the dynamic re-planning stage. Currently, only path lengths and the immediate next collision state is checked before sending the next poses to the robots. This does not check for additional situations, such as if the robot would get stuck in a loop, or stays at the same pose for multiple calls of compute_next_waypoint() until it finds a shorter (but not necessarily better) path. Further, with more number of robots, the number of particles would have to be increased, to avoid the chances of particles getting exhausted during neighbor search. But this would come with a cost of computation.

Finally, the computations for YARM only take place when all robots arrive at their immediate next waypoint, but no computations currently occur during the robot motion. This was deliberately done to observe waypoint computation, which means that some robots would have to wait until other robots reach their next waypoints. For a large number of robots, this could lead to a larger total wait time.

8.2 Future Work

With the success of the algorithm planning paths for the 2M1Ma, 1M1Ma and 3M case, the next step would be to scale the system by adding more mobile robots and manipulators and observing the performance. The additional complexity with each robot would mean that pipelines for computation would have to be optimized and computational resources organized. One of the areas to help improve the computational efficiency would be researching better path selection criteria during dynamic re-planning. Due to the coupled nature of this step with the number of particles used, this is an interesting but vast research topic in itself, which could have potential uses to other kinds of research. It was observed that the random sampling of particles using Python's **random**

library, often lead to empty pockets in space. A means to achieve better particle distribution during random sampling, is another area to explore. This could be coupled with exploring how to dynamically tune the number of particles depending on the sparseness of the environment. Such dynamic tuning would potentially speed up the path planning process as there is a smaller space of particles to search, and this could allow for searching more configurations of the robot if required. Thus, there is scope for improvement in many elements of the algorithm.

Other scope of research would lie in practical implementation of the algorithm on real hardware, to observe real time performance. Of course, this doesn't eliminate the need for better distribution of compute power; but will come with its own challenges, such as fixing communication delays between various elements of the system. Finally, the system could be tested with humans in the environment, to understand how it handles uncertainty. Behavioral prediction algorithms could be coupled with the dynamic re-planning stage to ensure that interactions with humans are safe. This would lead to useful multi-agent collaborative robotics systems in industrial environments.

BIBLIOGRAPHY

[1] Dubanchet V., Saussié D., Alazard D., Bérard C., Le Peuvédic C. (2015), "*Motion Planning and Control of a Space Robot to Capture a Tumbling Debris"*. In: Bordeneuve-Guibé J., Drouin A., Roos C. (eds) Advances in Aerospace Guidance, Navigation and Control. Springer, Cham. https://doi.org/10.1007/978-3-319-17518-8_40

[2]https://www.esa.int/Safety_Security/Clean_Space/ESA_commissions_world_s_first_space_debris_removal

[3] Bogue, R. (2019), "*Robots in recycling and disassembly*", Industrial Robot, Vol. 46 No. 4, pp. 461-466. https://doi.org/10.1108/IR-03-2019-0053

[4] Bakari MJ, Zied KM, Seward DW. "*Development of a Multi-Arm Mobile Robot for Nuclear Decommissioning Tasks*". International Journal of Advanced Robotic Systems. December 2007. doi:10.5772/5665

[5] R. Bischoff, U. Huggenberger and E. Prassler, "*KUKA youBot - a mobile manipulator for research and education*," 2011 IEEE International Conference on Robotics and Automation, Shanghai, 2011, pp. 1-4, doi: 10.1109/ICRA.2011.5980575.

[6] T.Jenkel, R.Kelley, P.Shepanski, "*Mobile Manipulation for the KUKA youBot platform*", WPI, 2013

[7] T. Mercy, W. Van Loock and G. Pipeleers, "*Real-time motion planning in the presence of moving obstacles*," 2016 European Control Conference (ECC), Aalborg, 2016, pp. 1586-1591, doi: 10.1109/ECC.2016.7810517.

[8] Hart, P., Nilsson, N., & Raphael, B. (1968). *"A Formal Basis for the Heuristic Determination of Minimum Cost Paths"*. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100–107. https://doi.org/10.1109/tssc.1968.300136

[9] S. M. LaValle. "*Rapidly-exploring random trees: A new tool for path planning*". TR 98-11, Computer Science Dept., Iowa State University, 1998.

[10] D. Harabor; A. Grastien (2011). "*Online Graph Pruning for Pathfinding on Grid Maps*". 25th National Conference on Artificial Intelligence. AAAI.

[11] Lydia Kavraki, Petr Svestka, Jean Latombe, and Mark Overmars. 1994. "*Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces*". Technical Report. Stanford University, Stanford, CA, USA.

[12] David Coleman, Ioan A. Șucan, Sachin Chitta, Nikolaus Correll, "*Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study*", Journal of Software Engineering for Robotics, 5(1):3–16, May 2014. doi: 10.6092/JOSER_2014_05_01_p3.

[13] LaValle SM, Kuffner JJ (2001) "*Randomized kinodynamic planning*". Int J Robot Res 20(5):378–400

[14] Amato, N. M., and Wu, Y. 1996. "*A randomized roadmap method for path and manipulation planning*". IEEE International Conference on Robotics and Automation, pp. 113–120 .

[15] Kang, G., Kim, Y.B., Lee, Y.H. et al. *"Sampling-based motion planning of manipulator with goal-oriented sampling"*. Intel Serv Robotics 12, 265–273 (2019). https://doi.org/10.1007/s11370-019-00281-y

[16] https://www.mathworks.com/help/nav/ug/motion-planning-with-rrt-for-manipulators.html

[17] Nikos Vlassis, *"A Concise Introduction to Multiagent Systems and Distributed Artificial Intelligence"*, Morgan & Claypool, 2007, doi: 10.2200/S00091ED1V01Y200705AIM002.

[18] Barraquand J., Kavraki L., Latombe JC., Li TY., Motwani R., Raghavan P. (1996) *"A Random Sampling Scheme for Path Planning*". In: Giralt G., Hirzinger G. (eds) Robotics Research. Springer, London. https://doi.org/10.1007/978-1-4471-1021-7_28

[19] J. S. Bellingham, M. Tillerson, M. Alighanbari and J. P. How, "*Cooperative path planning for multiple UAVs in dynamic and uncertain environments*," Proceedings of the 41st IEEE Conference on Decision and Control, 2002., Las Vegas, NV, USA, 2002, pp. 2816-2822 vol.3, doi: 10.1109/CDC.2002.1184270.

[20] Y. Wu, S. Wu and X. Hu, "*Cooperative path planning of UAVs & UGVs for a persistent surveillance task in urban environments*," in IEEE Internet of Things Journal, doi: 10.1109/JIOT.2020.3030240.

[21] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard: *"Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters"*, IEEE Transactions on Robotics, Volume 23, pages 34-46, 2007

[22] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard: *"Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling"*, In Proc. of the IEEE International Conference on Robotics and Automation (ICRA), 2005

[23] https://depts.washington.edu/barc/projects/sanding-assist-system

[24] C. Devine, *"Material removal rate control for a teleoperated robotic sander"*, Masters Thesis, University of Washington, 2018

[25] Stanford Artificial Intelligence Laboratory et al. (2018). *Robotic Operating System*. Retrieved from https://www.ros.org

[26] Liu, Jun S.; Chen, Rong (1998-09-01). "*Sequential Monte Carlo Methods for Dynamic Systems*". Journal of the American Statistical Association. 93 (443): 1032–1044. doi:10.1080/01621459.1998.10473765. ISSN 0162-1459.

[27] Julier, S., & Uhlmann, J. (2004). "*Unscented filtering and nonlinear estimation*". Proceedings of the IEEE, 92, 401-422.

[28] Foote, Tully. (2013). *"Tf: The transform library"*. IEEE Conference on Technologies for Practical Robot Applications, TePRA. 1-6. 10.1109/TePRA.2013.6556373.

[29] Koenig, N., & Howard, A. (2006). *Gazebo-3d multiple robot simulator with dynamics*.

[30] Hershberger, D., Gossow, D., & Faust, J. (2019). *RViz, 3D visualization tool for ROS*. URL: http://wiki. ros. org/rviz [cited 06-08-2018].

[31] R. B. Rusu and S. Cousins, "*3D is here: Point Cloud Library (PCL),*" 2011 IEEE International Conference on Robotics and Automation, Shanghai, 2011, pp. 1-4, doi: 10.1109/ICRA.2011.5980567.

[32] Nguyen, Anh & Le, Bac. (2013). "*3D point cloud segmentation: A survey*". IEEE Conference on Robotics, Automation and Mechatronics, RAM - Proceedings. 225-230. 10.1109/RAM.2013.6758588.

[33] Rizzo, A., Haustein, J. A., & Bianchi, C. "*Extracting contact surfaces from point-cloud data for autonomous placing of rigid objects"*, Thesis, 2020. URL: http://webthesis.biblio.polito.it/id/eprint/14410

[34] E. G. Gilbert, D. W. Johnson and S. S. Keerthi, "*A fast procedure for computing the distance between complex objects in three-dimensional space*," in IEEE Journal on Robotics and Automation, vol. 4, no. 2, pp. 193-203, April 1988, doi: 10.1109/56.2083.

[35] Jaillet, Léonard & Siméon, Thierry. (2004). "*A PRM-based motion planner for dynamically changing environments".* 1606 - 1611 vol.2. 10.1109/IROS.2004.1389625.

[36] C. Fragkopoulos and A. Gräser, "*Sampling Based Path Planning for High DoF Manipulators without Goal Configuration*", The 18th World Congress of the International Federation of Automatic Control (IFAC 2011), pp. 11568-11573, 2011

[37] Barber, C. B.; Dobkin, D. P.; and Huhdanpaa, H. T. "*The Quickhull Algorithm for Convex Hulls*." ACM Trans. Mathematical Software 22, 469-483, 1996.

APPENDIX A

Note: The code provided here is part of a larger repository, and only reflects the task of immediate interest. For reference, the repository can be accessed at: https://github.com/ankurjay/ankurbot/tree/multi_processing_trials_skidsteer_astar

Relative paths are provided per this repository, for reference.

Python Code for TaskQueue, CostQueue and Information Classes

Path: ankurjay/ankurbot/ankurbot_description/scripts/taskqueue.py

```
import math

class TaskQueue:
    def __init__(self):
          self.robots = {} # This stores the robots
          self.debris = {} # This stores the debris
          self.targets = {} # This stores the targets
          self.idle_robots = [] # This is a queue of idle robots
          self.queue = [] # Queue will either be filled with all robots and some debris, or all debris and some robots

    def processOnce(self):
          """
          Assigns the debris or target to the nearest robot.
          """

          dbr = self.debris.items()
          tgt = self.targets.items()

          if len(dbr) == 0:
                    pass
          else:

                    # Search in queue if target has already been assigned to some robot
                    assigned_targets = [item[1] for item in self.queue]
                    for pair in dbr:

                              if pair in assigned_targets:
                                        # If the pair is already assigned a robot, don't do anything
                                        pass

                              else:
                                        # The pair is not assigned a robot. So look for an idle robot.
                                        mindist = 999999
                                        for rbt in self.idle_robots:
                                                  if self.distance(rbt[1], pair[1]) < mindist:
                                                            mindist = self.distance(rbt[1], pair[1])
                                                            temp = (pair, rbt)

                                        dbr.remove(temp[0])
                                        self.queue.append((temp[1][0], temp[0][0], "Debris", temp[0][1]))
                                        del self.debris[temp[0][0]]
                                        self.idle_robots.remove(temp[1])


          if len(tgt) == 0:
                    pass
          else:

                    # Search in queue if target has already been assigned to some robot
                    assigned_targets = [item[1] for item in self.queue]
                    for pair in tgt:
```

```
                        if pair in assigned_targets:
                                # If the pair is already assigned a robot, don't do anything
                                pass

                        else:
                                # The pair is not assigned a robot. So look for an idle robot.
                                mindist = 999999
                                for rbt in self.idle_robots:
                                        if self.distance(rbt[1], pair[1]) < mindist:
                                                mindist = self.distance(rbt[1], pair[1])
                                                temp = (pair, rbt)

                                tgt.remove(temp[0])
                                self.queue.append((temp[1][0], temp[0][0], "Target", temp[0][1]))
                                del self.targets[temp[0][0]]
                                self.idle_robots.remove(temp[1])


        # If debris appears, I cannot do dynamic removal. THIS IS IT.
        #print("\n&&&& Finally, TaskQueue is like this: ")
        #print("\n&&&& Idle Robots : ", self.idle_robots)
        #print("\n&&&& Non-Assigned Targets and Debris : ", self.targets, self.debris)
        #print("\n&&&& Queue : ", self.queue)

    def updateOnce(self, obj_dict, dtype):
        """
        Take the current dictionary of robots, update their poses and make them idle. Or update the debris and targets.
        """
        if dtype == "ROBOT":
                for robot_name, data in obj_dict.items():
                        if robot_name not in [item[0] for item in self.queue]:                      # If robot is not
in processing queue,
                                if robot_name not in [item[0] for item in self.idle_robots]:     # AND if robot is
not in idle list
                                        self.robots[robot_name] = data["Pose"]
        # update pose and add to idle list
                                        self.idle_robots.append([robot_name, data["Pose"]])
                                else:
                                        # If robot is already in idle list, it may have moved to accomodate other
robots.
                                        idx = [item[0] for item in self.idle_robots].index(robot_name)

                                        self.robots[robot_name] = data["Pose"]
        # Then update pose of the robot in the dictionary as well as the idle_robots queue
                                        self.idle_robots[idx][1] = data["Pose"]
                        # If robot is in processing queue, does not update the robot
                        # So we must pop robot from queue then do updateOnce then it will be added to idle list
        elif dtype == "DEBRIS":
                #print("\n^^^UpdateOnce is called and the dictionary input is : ", obj_dict.keys())

                templist = list(obj_dict.items())
                #print("\n]]] Currently the list of debris available is : ", templist)
                current_tgts = [item[1] for item in self.queue]
                for pair in templist:
        # If target is not being currently approached, add it to the dict of targets
                        if pair[0] not in current_tgts:
                                print("\n^^^Adding ",pair[0]," to list of debris in TaskQueue data structure.")
                                self.debris[pair[0]] = pair[1]["Pose"]
        # update pose

        elif dtype == "TARGET":

                #print("\n^^^UpdateOnce is called and the dictionary input is : ", obj_dict.keys())

                templist = list(obj_dict.items())
                #print("\n]]] Currently the list of targets available is : ", templist)
                current_tgts = [item[1] for item in self.queue]
                for pair in templist:
        # If target is not being currently approached, add it to the dict of targets
                        if pair[0] not in current_tgts:
                                print("\n^^^Adding ",pair[0]," to list of targets in TaskQueue data structure.")
                                self.targets[pair[0]] = pair[1]["Pose"]
        # update pose


    def distance(self, p1, p2, bias = [1,1,0]):
        """Distance Heuristic function with bias along X and Y"""
```

```
          return math.sqrt(bias[0]*(p1.transform.translation.x - p2.transform.translation.x)**2 +
bias[1]*(p1.transform.translation.y - p2.transform.translation.y)**2 + bias[2]*(p1.transform.translation.z -
p2.transform.translation.z)**2)

    def getQueue(self):
          """Returns the queue so we can scan through it, find the index we want to delete, and send to popQueue()
separately"""
          return self.queue

    def deleteTargetFromQueue(self, target_name):
          """Find the target to delete"""
          index = [item[1] == target_name for item in self.queue].index(True)
          robot_target_pair = self.popQueue(index)
          print("\n???? ",target_name," was deleted from the Queue. Queue is now : ",self.queue)

    def popQueue(self, idx):
          """Pops the element from the queue at index idx"""
          return self.queue.pop(idx)


class Information:
    def __init__(self):
          self.gx = 0                                                    # These are grid parameters
          self.gy = 0
          self.gz = 0
          self.xmax = 0
          self.xmin = 0
          self.ymax = 0
          self.ymin = 0
          self.zmax = 0
          self.zmin = 0
          self.resolution_x = 0
          self.resolution_y = 0
          self.resolution_z = 0
          self.grid = None


class CostQueue:
    def __init__(self):
          self.queue = []

    def push(self, element):
          """Element must be a tuple consisting of list of COST first, then other stuff."""
          cost = element[0] + len(element[1])
          self.queue.append((cost, element[1]))
          self.queue = sorted(self.queue, key = lambda x : x[0])

    def popQueue(self):
          """Gets the most optimal element"""
          return self.queue.pop(0)

    def view(self):
          return self.queue[0]

    def isEmpty(self):
          return len(self.queue) == 0



    def printAgenda(self):
          """Highly specific method"""
          for element in self.queue:
                    templist = []
                    for dictionary in element[1]:
                              templist.append(dictionary["CurrentSeed"])
                    print("Cost : ",element[0], ", Agenda : ", templist)
```

Python Code for Commander

Path: ankurjay/ankurbot/ankurbot_description/scripts/commander.py

```
import roslib
roslib.load_manifest('ankurbot_multisim') #Load the package.xml of the relevant package
import rospy
import sys # Needed for main function argument sys.argv
import numpy as np
import math
import random
from ankurbot_multisim.msg import MapMetaData, GridIndex, GridIndexVector, MoveRobotAction,MoveRobotGoal
from ankurbot_multisim.srv import StateChange, StateChangeResponse
from ankurbot_multisim.srv import SpawnObstacles, SpawnObstaclesRequest
from std_msgs.msg import Int8
from geometry_msgs.msg import Pose, TransformStamped, Point, PoseStamped
import actionlib
from tf.transformations import euler_from_quaternion, quaternion_from_euler
from visualization_msgs.msg import Marker
from scipy.spatial import Delaunay
import copy

from taskqueue import *               # Required import for using TaskQueue class
from transforms import *
from markerpublisher import *


class Commander:
    def __init__(self, grid_threshold = 5):
          self.map_sub = rospy.Subscriber("planning_metadata", MapMetaData, self.callback)
          self.static_assoc_switch_pub = rospy.Publisher("static_association_switch", Int8, queue_size = 5)
          self.state_change_server = rospy.Service("commander_state", StateChange, self.state_change)
          self.obstacle_deletion = rospy.ServiceProxy("delete_obstacles", SpawnObstacles, persistent=True)
          self.masks_pub = rospy.Publisher("maps_and_masks", Marker, queue_size=1)
          self.particles_pub = rospy.Publisher("particles_and_waypoints",Marker,queue_size=1)


          self.current_state = 0                        # This is the current state of the state machine
          self.static_association = False     # This is the state of static association

          self.information = Information()    # This stores the grid information

          self.tasklist = TaskQueue()                   # This is the task processing queue
          self.threshold = grid_threshold      # This is the threshold for search

          self.objects = {}
          self.action_clients = {}
          self.search_space = {}
          self.private_spaces = {}
          self.grid_velocities = {}
          self.waypoints = {}
          self.recent = {}
          self.colors = {}
          self.clist = [(1,0,0,1), (0,1,0,1), (0,0,1,1)] # Add more colors here are required

          self.forbiddenzone = []


    def callback(self, data):
          """
          This is the root function that gets called whenever new data is available.
          We will initialize the grid, then extract objects, then publish the static association switch
          """
          self.initialize_grid(data)
          self.extract_objects(data)
          self.publish_static_association()

    def initialize_grid(self, data):
          """
          This function simply initializes self.grid and sets the grid parameter values
          """
```

```
        self.information.grid = np.reshape(data.grid, ((data.gx, data.gy, data.gz))) # This already contains the
occupancies as 0,1,2
        self.information.gx = data.gx
        self.information.gy = data.gy
        self.information.gz = data.gz
        self.information.xmax = data.xmax
        self.information.xmin = data.xmin
        self.information.ymax = data.ymax
        self.information.ymin = data.ymin
        self.information.zmax = data.zmax
        self.information.zmin = data.zmin
        self.information.resolution_x = (self.information.xmax - self.information.xmin)/self.information.gx;
        self.information.resolution_y = (self.information.ymax - self.information.ymin)/self.information.gy;
        self.information.resolution_z = (self.information.zmax - self.information.zmin)/self.information.gz;

    def extract_objects(self, data):
        """
        This function extracts the objects from data sent by database.py, and sorts them into 4 main categories.
        It also initializes the search spaces and starts the action clients for each robot
        """

        for idx in range(len(data.frames)):                                             # Update pose and mask
details
                if data.objects[idx] not in self.objects.keys():    # If key does not exist
                        self.objects[data.objects[idx]] = {}                    # Initialize empty dictionary
                self.objects[data.objects[idx]][data.frames[idx]] = {\

                "Pose" : data.seed_poses[idx], \

                "Seed" : find_grid_idx(data.seed_poses[idx], self.information), \

                "Grid" : find_grid_idx(data.grid_indices[idx], self.information), \

                "Mask" : None, \

                }
                self.objects[data.objects[idx]][data.frames[idx]]["Mask"] =
make_binary_mask(self.objects[data.objects[idx]][data.frames[idx]]["Grid"], self.information.grid)
                #temp = self.objects[data.objects[idx]][data.frames[idx]]["Mask"]
                #print(data.frames[idx], np.sum(temp))

        for robot in self.objects["Mobile"].keys():                                     # Initialize action clients and
blank out the search spaces only once
                if "Mobile" not in self.action_clients.keys():
                        self.action_clients["Mobile"] = {}
                        self.search_space["Mobile"] = {}
                        self.grid_velocities["Mobile"] = {}
                        self.colors["Mobile"] = {}
                elif robot not in self.action_clients["Mobile"].keys():
                        self.action_clients["Mobile"][robot] = actionlib.SimpleActionClient(robot+"/move_robot",
MoveRobotAction)
                        self.search_space["Mobile"][robot] = ""
                        self.grid_velocities["Mobile"][robot] = (0,0,0,0,0,0)
                        self.colors["Mobile"][robot] = self.clist.pop(0)

        if "Manipulator" in self.objects.keys():
                for robot in self.objects["Manipulator"].keys():        # Initialize action clients and blank out the
search spaces only once
                        if "Manipulator" not in self.action_clients.keys():
                                self.action_clients["Manipulator"] = {}
                                self.search_space["Manipulator"] = {}
                                self.grid_velocities["Manipulator"] = {}
                        elif robot not in self.action_clients["Manipulator"].keys():
                                self.action_clients["Manipulator"][robot] =
actionlib.SimpleActionClient(robot+"/move_robot", MoveRobotAction)
                                self.search_space["Manipulator"][robot] = ""
                                self.grid_velocities["Manipulator"][robot] = (0,0,0,0,0,0)

        if "Obstacle" in self.objects.keys():
                self.forbiddenzone = []
                for obstacle in self.objects["Obstacle"].keys():
                        gidxs = self.objects["Obstacle"][obstacle]["Grid"]
                        gidz = [(item[0], item[1]) for item in gidxs]
                        self.private_spaces[obstacle] = gidz
                        self.forbiddenzone.extend(gidz)

    def publish_static_association(self):
```

```
            """This function simply publishes the current state of static association"""
            if self.static_association == False:
                    self.static_assoc_switch_pub.publish(0)
            elif self.static_association == True:
                    self.static_assoc_switch_pub.publish(1)

    def get_current_state(self):
            """Getter function to get current state"""
            return self.current_state

    def set_current_state(self, state):
            """Setter function to set current state"""
            self.current_state = state

    def state_change(self,req):
            """
            Function to carry out state changes in the state machine
            This is executed via service calls, so it must return a flag of True or False
            """
            flag = False
            if self.get_current_state() == 0:
                    """Initial State is 0, trigger anything to go to state 1"""
                    flag = self.state_1()

            elif self.get_current_state() == 1:
                    """
                    Current State is 1
                    1) Stay on Current State
                    2) Go to State 2, where you do static association
                    """
                    if (req.next_state, req.current_state, req.previous_state, req.jump_state) == (False, True, False,
0):
                            flag = self.state_1()
                    elif (req.next_state, req.current_state, req.previous_state, req.jump_state) == (True, False, False,
0):
                            flag = self.state_2()
                    else:
                            pass

            elif self.get_current_state() == 2:
                    """
                    Current State is 2
                    1) Stay in current state and redo static association
                    2) Skip to State 4, Non-Collab mode
                    3) Skip to State 5, Collab mode
                    4) Skip to State 3, Intermediate Choosing Stage
                    """
                    if (req.next_state, req.current_state, req.previous_state, req.jump_state) == (False, True, False,
0): # Stay in current state, redo static assoc.
                            self.static_association = False
                            self.static_assoc_switch_pub.publish(0)
                            rospy.sleep(2)
                            flag = self.state_2()
                    elif (req.next_state, req.current_state, req.previous_state, req.jump_state) == (False, False, False,
4): # Move to bounded state directly
                            flag = self.state_4()
                    elif (req.next_state, req.current_state, req.previous_state, req.jump_state) == (False, False, False,
5): # Move to unbounded state directly
                            flag = self.state_5()
                    elif (req.next_state, req.current_state, req.previous_state, req.jump_state) == (True, False, False,
0): # Move to next state to choose bounded or unbounded
                            flag = self.state_3()
                    else:
                            pass

            elif self.get_current_state() == 3:
                    """
                    Current State is 3
                    1) Stay in current state and do nothing
                    2) Skip to State 4, Non-Collab mode
                    3) Skip to State 5, Collab mode
                    4) Skip to State 2, and redo static association
                    """

                    if (req.next_state, req.current_state, req.previous_state, req.jump_state) == (False, True, True, 0):
# Stay in current state
                            flag = self.state_3()
```

```
                elif (req.next_state, req.current_state, req.previous_state, req.jump_state) == (False, False, False,
4): # Move to bounded state directly
                        flag = self.state_4()
                elif (req.next_state, req.current_state, req.previous_state, req.jump_state) == (False, False, False,
5): # Move to unbounded state directly
                        flag = self.state_5()
                elif (req.next_state, req.current_state, req.previous_state, req.jump_state) == (False, False, True,
0): # Move to previous state to redo static assoc
                        self.static_association = False
                        self.static_assoc_switch_pub.publish(0)
                        rospy.sleep(2)
                        flag = self.state_2()
                else:
                        pass

        elif self.get_current_state() == 4:
                """
                Current State is 4, Non-Collab Mode
                1) Stay in current state, recheck queue for debris or targets and do tasks
                2) Skip to State 6, Cleanup mode
                3) Skip to State 3, Intermediate Choosing Stage
                """
                if (req.next_state, req.current_state, req.previous_state, req.jump_state) == (False, True, False,
0): # Stay on current state
                        flag = self.state_4()
                elif (req.next_state, req.current_state, req.previous_state, req.jump_state) == (True, False, False,
0): # Move to cleanup state
                        flag = self.state_6()
                elif (req.next_state, req.current_state, req.previous_state, req.jump_state) == (False, False, True,
0): # Move to previous state and choose between bounded and unbounded or redo static
                        flag = self.state_3()
                else:
                        pass

        elif self.get_current_state() == 5:
                """
                Current State is 5
                1) Stay in current state, recheck queue for debris and targets and do tasks
                2) Skip to State 6, Cleanup mode
                3) Skip to State 3, Intermediate Choosing Stage
                """
                if (req.next_state, req.current_state, req.previous_state, req.jump_state) == (False, True, False,
0): # Stay on current state
                        flag = self.state_5()
                elif (req.next_state, req.current_state, req.previous_state, req.jump_state) == (True, False, False,
0): # Move to cleanup state
                        flag = self.state_6()
                elif (req.next_state, req.current_state, req.previous_state, req.jump_state) == (False, False, True,
0): # Move to previous state and choose between bounded and unbounded or redo static
                        flag = self.state_3()
                else:
                        pass

        elif self.get_current_state() == 6:
                """
                Current State is 6, Cleanup has been completed.
                Reset state to 0
                """
                flag = self.reset_to_state_0()

        if flag == True:
                return StateChangeResponse(True, "Request processed successfully. Current State is
"+str(self.current_state))
        else:
                return StateChangeResponse(False, "Request failed. Current State is "+str(self.current_state))

    def state_1(self):
        self.set_current_state(1)
        print("\n<STATE 1> System Started.\
                        \nTrigger next_state (1,0,0,0) to continue requesting Static Association. \
                        \nTrigger current_state (0,1,0,0) to pause\n")
        self.static_association = False
        return True

    def state_2(self):
        self.set_current_state(2)
        self.static_association = True
```

```
            print("\n<STATE 2> Static Association Saved.")

            waiting_list = []
            done_list = []

            for robot_type in self.action_clients.keys():
                    for robot_name in self.action_clients[robot_type].keys():
                            self.action_clients[robot_type][robot_name].wait_for_server()

                            goal = MoveRobotGoal()
                            goal.target = Pose()
                            goal.action = "CALIBRATION"
                            self.action_clients[robot_type][robot_name].send_goal(goal, done_cb = None, active_cb =
None, feedback_cb = None)

                            # Send the goals and wait only for 1 second. If state is not succeeded, add to waiting list
                            self.action_clients[robot_type][robot_name].wait_for_result(rospy.Duration.from_sec(1.0))

                            if self.action_clients[robot_type][robot_name].get_state() != 3:
                                    print("For ",robot_name, "the state is ",
self.action_clients[robot_type][robot_name].get_state())
                                    waiting_list.append((robot_type, robot_name))

                            elif self.action_clients[robot_type][robot_name].get_state() == 3:
                                    print("For ",robot_name, "the state is ",
self.action_clients[robot_type][robot_name].get_state())
                                    done_list.append((robot_type, robot_name))
                                    result = self.action_clients[robot_type][robot_name].get_result()
                                    # result.message will be in the form SPACE=XXX/AREA=XX,XX,XX,XX,XX,XX

                                    # Update the search space for each robot, and update private_spaces
                                    self.search_space[robot_type][robot_name] =
result.message.split('/')[0].split('=')[-1]
                                    if result.message.split('/')[1].split('=')[-1] != "NONE":
# If there is an AREA mentioned
                                            temp = result.message.split('/')[1].split('=')[-1].split(',')
                                            self.private_spaces[robot_name] = tuple([int(val) for val in temp])


            while len(waiting_list) > 0: # Some robots, mainly HC10, will be in the waiting list during calibration
                    print("\nWAITING FOR ", waiting_list, " TO CALIBRATE. DO NOT CHANGE PRE-EMPT OR CHANGE STATE.")
                    for items in waiting_list:
                            self.action_clients[items[0]][items[1]].wait_for_result(rospy.Duration.from_sec(1.0))
                            if self.action_clients[items[0]][items[1]].get_state() != 3:
                                    pass
                            elif self.action_clients[items[0]][items[1]].get_state() == 3:
                                    done_list.append(items)
                                    waiting_list.remove(items)
                                    result = self.action_clients[robot_type][robot_name].get_result()

                                    # Update the search space for each robot, and update private_spaces
                                    self.search_space[robot_type][robot_name] =
result.message.split('/')[0].split('=')[-1]
                                    # if result.message.split('/')[1].split('=')[-1] != "NONE":
                                    #     temp = result.message.split('/')[1].split('=')[-1].split(',')
                                    #     self.private_spaces.append(tuple([int(val) for val in temp]))


            print("Robots Controls Identified. "\
                    "Trigger jump_state (0,0,0,4) to continue to bounded mode, "\
                    "jump_state (0,0,0,5) to go to unbounded mode, "\
                    "next_state (1,0,0,0) to intermediate state to choose, "\
                    "current_state (0,1,0,0) to redo Static Association")
            return True

    def state_3(self):
            self.set_current_state(3)
            print("\n<STATE 3> Intermediate Stage.\
                            \nTrigger jump_state (0,0,0,4) to continue to bounded mode, \
                            \njump_state (0,0,0,5) to continue to unbounded mode, \
                            \nprevious_state (0,0,1,0) to redo static association")
            return True

    def state_4(self):
            self.set_current_state(4)
            print("\n<STATE 4> Mobile Robots carrying out scheduled tasks within boundaries.")
            self.process_targets("Bounded")
```

```
            print("\nAll tasks completed for the moment. \
                                \nTrigger next_state (1,0,0,0) to continue to cleanup mode, \
                                \ncurrent_state (0,1,0,0) to stay bounded and check for more tasks, \
                                \nprevious_state (0,0,1,0) to choose between bounded and unbounded mode")
            return True

    def state_5(self):
            self.set_current_state(5)
            print("\n<STATE 5> Collaboration mode is active. Mobile Robots will carry out scheduled tasks interactively
with Manipulators.")
            self.process_targets("Boundless")
            print("\nAll tasks completed for the moment. \
                                \nTrigger next_state (1,0,0,0) to continue to cleanup mode, \
                                \ncurrent_state (0,1,0,0) to stay unbounded and check for more tasks, \
                                \nprevious_state (0,0,1,0) to choose between bonded and unbounded mode")
            return True

    def state_6(self):
            self.set_current_state(6)
            print("\n<STATE 6> Cleanup in progress...")
            self.cleanup()
            print("\nCleanup Complete. Trigger anything to Finish.")
            return True

    def reset_to_state_0(self):
            self.set_current_state(0)
            print("\nSystem restarted. ")
            return True

    def process_targets(self, mode="Boundless"):

            # 1) Update the tasklist once before starting the process
            self.tasklist.updateOnce(self.objects["Mobile"], "ROBOT")
            if "Target" in self.objects.keys():
                    self.tasklist.updateOnce(self.objects["Target"], "TARGET")
            if "Debris" in self.objects.keys():
                    self.tasklist.updateOnce(self.objects["Debris"], "DEBRIS")
            self.tasklist.processOnce()
            queue = self.tasklist.getQueue()                                                                    # Queue contains
(mobilerobotname, targetname, targettype, targetpose)
            print("\nFirst time Queue Processed : ",[(item[0],item[1],item[2]) for item in queue])

            # 2) Initialize an empty waiting list
            waiting_list = []
            print("\nInitialized Waiting List to EMPTY : ", waiting_list)


            ct = False
            # 3) Keep iterating as long as the queue has mobile robots in it or waiting list has elements in it
            while ((len(queue) > 0 or len(waiting_list) > 0) and ct is True) or ((len(queue) > 0 or len(waiting_list) >= 0)
and ct is False):
                    print("\nIn process_targets() while loop. Queue length = ",len(queue))

                    # 3-1) Update the tasklist to check if there are robots still assigned to tasks, so that the queue is
non-empty.
                    self.tasklist.updateOnce(self.objects["Mobile"], "ROBOT")
                    if "Target" in self.objects.keys():

                            temp = self.objects["Target"]

                            self.tasklist.updateOnce(temp, "TARGET")

                    if "Debris" in self.objects.keys():

                            temp = self.objects["Debris"]

                            self.tasklist.updateOnce(temp, "DEBRIS")

                    self.tasklist.processOnce()
                    queue = self.tasklist.getQueue()                                                            # Receive the
current processing queue
                    print("\nReprocessed queue inside process_targets() while loop. ", \
                                    [(q[0],q[1],q[2], \
                                    (q[3].transform.translation.x,
                                    q[3].transform.translation.y,
                                    q[3].transform.translation.z)) for q in queue], " and its length is ",len(queue))
```

```
                    # 3-2) Parse the processing queue and get the next waypoint only, for each mobile robot in the queue.
This will set or update self.waypoints.
                    self.compute_next_waypoint(queue)
                    print("\nself.compute_next_waypoint() is complete.")

                    # 3-3) Publish the waypoints and let ALL robots reach their targets
                    for robot_type in self.action_clients.keys():
                             for robot_name in self.action_clients[robot_type].keys():
                                      self.action_clients[robot_type][robot_name].wait_for_server()

                                      goal = MoveRobotGoal()

                                      # 3-3-1) If the robot is a Mobile robot and it has been assigned waypoints due to
self.compute_next_waypoint(), send the robot to its NextPose
                                      if robot_type == "Mobile" and robot_name in self.waypoints.keys():
                                                        # If the robot is a mobile robot
                                               temptarget = self.waypoints[robot_name]["NextPose"]     # This is a
Pose()
                                               if isinstance(temptarget, Pose):
                                                        temptarget.orientation =
angle_correction(temptarget.orientation, 90)
                                                        #temptarget.orientation =
angle_correction(temptarget.orientation, 90)
                                                        goal.target = temptarget
                                               elif isinstance(temptarget, TransformStamped):
                                                        temp = Pose()
                                                        temp.position.x = temptarget.transform.translation.x
                                                        temp.position.y = temptarget.transform.translation.y
                                                        temp.position.z = temptarget.transform.translation.z
                                                        temp.orientation.x = temptarget.transform.rotation.x
                                                        temp.orientation.y = temptarget.transform.rotation.y
                                                        temp.orientation.z = temptarget.transform.rotation.z
                                                        temp.orientation.w = temptarget.transform.rotation.w
                                                        temp.orientation = angle_correction(temp.orientation, 90)
                                                        #temp.orientation = angle_correction(temp.orientation, 90)
                                                        goal.target = temp


                                               print("\nAction being sent to the robot ",robot_name," is goaltarget : ",
goal.target)
                                               goal.action = "False"
                                      # Rotate about End-Effector for now
                                               self.action_clients[robot_type][robot_name].send_goal(goal, done_cb =
None, active_cb = None, feedback_cb = self.update_velocities)
                                               waiting_list.append((robot_type, robot_name))
# Send commands for all robots to start their tasks, ONCE.
                                               print("\nAdded ",robot_name," to waiting list. WL is now : ",
waiting_list)



                                      # 3-3-2) Otherwise, if the robot is a Manipulator, just check for collisions
                                      elif robot_type == "Manipulator":
                    # If the robot is a manipulator
                                               if (robot_type, robot_name) not in waiting_list:                  # If the
manipulator is not in the waiting list, then send a new goal
                                                        print("\n",robot_name," is not in the waiting list.")
                                                        if not
self.check_expected_collisions_in_manipulators(robot_name): # There is no collision, then this returns False
                                                                 goal.target = Pose()
                                                                 goal.action = "ON"
                                      # Start the manipulator machining

self.action_clients[robot_type][robot_name].send_goal(goal, done_cb = None, active_cb = None, feedback_cb =
self.update_velocities)
                                                                 waiting_list.append((robot_type, robot_name))
# Send commands for all robots to start their tasks, ONCE.
                                                        else:
                                                                 goal.target = Pose()
                                                                 goal.action = "OFF"
                                      # Start the manipulator machining

self.action_clients[robot_type][robot_name].send_goal(goal, done_cb = None, active_cb = None, feedback_cb =
self.update_velocities)
                                                                 waiting_list.append((robot_type, robot_name))     #
Send commands for all robots to start their tasks, ONCE.
                                                        print("\nAdded ",robot_name," to waiting list. WL is now : ",
waiting_list)
```

```
                                        else:
                                        # Otherwise if the manipulator is in the waiting list,
                                                print("\n",robot_name," is in the waiting list.")
                                                if not
self.check_expected_collisions_in_manipulators(robot_name): # There is no collision, then this returns False
                                                        goal.target = Pose()
                                                        goal.action = "ON"
                                # Start the manipulator machining

self.action_clients[robot_type][robot_name].send_goal(goal, done_cb = None, active_cb = None, feedback_cb =
self.update_velocities)
                                                else:
                                                        goal.target = Pose()
                                                        goal.action = "OFF"
                        # Start the manipulator machining

self.action_clients[robot_type][robot_name].send_goal(goal, done_cb = None, active_cb = None, feedback_cb =
self.update_velocities)
                                                print("\n",robot_name, " is already in waiting list. WL is now :
", waiting_list)
                ct = True

                # 3-4) Now all robots have been sent their action signals and are in the waiting list. So until all
robots in waiting list have reached their next waypoints
                while len(waiting_list) > 0:

                        for items in waiting_list:


self.action_clients[items[0]][items[1]].wait_for_result(rospy.Duration.from_sec(0.05))    # Wait for result

                                # 3-4-1) If the robot on the waiting list has not reached its current waypoint, or
the manipulator has not finished operation
                                if self.action_clients[items[0]][items[1]].get_state() != 3:
                                # Check If action is going on
                                        #print("\nAction is still going on for ",items[1])

                                        # 3-4-1-1) If the item on waiting list is a manipulator, just check for
collisions
                                        if items[0] == "Manipulator":
                                                                # If the item is a manipulator
                                                if self.check_expected_collisions_in_manipulators(items[1]):
                                # Check if the manipulator must be paused or restarted
                                                        print("\nExpected Collision, Turning OFF")
                                                        goal.target = Pose()
                                                        goal.action = "OFF"
                                # Start the manipulator machining

self.action_clients[robot_type][robot_name].send_goal(goal, done_cb = None, active_cb = None, feedback_cb =
self.update_velocities)
                                                else:
                                                        print("\nNo Expected Collision, Turning ON")
                                                        goal.target = Pose()
                                                        goal.action = "ON"
                                # Start the manipulator machining

self.action_clients[robot_type][robot_name].send_goal(goal, done_cb = None, active_cb = None, feedback_cb =
self.update_velocities)
                                        elif items[0] == "Mobile":

          # If the item is a mobile robot
                                                #print("\nAction is still going on for : ",items[1]," and
waiting list is ", waiting_list)
                                        pass
#=========================================================================# No Rerouting implemented for now



                                # 3-4-2) Otherwise, if the robot on the waiting list has reached its current
waypoint, or the manipulator has finished operation
                                elif self.action_clients[items[0]][items[1]].get_state() == 3:
                                # Check If action is completed, i.e. target is reached
                                        print("\nCurrent Action is completed for ", items[1])

                                        # 3-4-2-1) If the robot is a Manipulator, remove it from the waiting list
                                        if items[0] == "Manipulator":
                                                waiting_list.remove(items)
                                                continue
```

```
                                        elif items[0] == "Mobile":
                                                check = self.waypoints[items[1]]["CurrentWaypoints"][1]

                                                tgt =
self.objects[self.waypoints[items[1]]["Type"]][self.waypoints[items[1]]["Target"]]["Seed"][0]

                                                waiting_list.remove(items)
                                                                                        # Remove robot from waiting
list
                                                print("\nRemoving robot from waiting list. Waiting list is now
", waiting_list)

                                                self.waypoints[items[1]]["NextPose"] = None

                                                print("\n****We have reached: ",check," and target is : ",tgt)
                                                if check[0] == tgt[0] and check[1] == tgt[1]:
                                                        # If the current waypoint was the goal
                                                        print("\nCurrent Waypoint was the goal. Deleting the
target.")

                                                        # if self.waypoints[items[1]]["Type"] == "Target":
                                                # If this was a Target


                                                        idx = [titem[0] for titem in queue].index(items[1])
                                                # Find the index of that robot in the queue
                                                        targetname, dtype = queue[idx][1], queue[idx][2]

                                                        self.tasklist.deleteTargetFromQueue(targetname)
                                                        # Remove the target from the tasklist
                                                        self.signal_target_deletion(targetname, dtype)
                                                        # Send a message to delete the target
                                                        queue = self.tasklist.getQueue()
                                                        self.waypoints.pop(items[1])
                                                                        # Remove the robot from the
waypoints dict

                                                        rospy.sleep(0.1)
                                                else:
                                                        print("\nWe have not reached the goal yet")

                                # Now check if all items in the waiting list are only manipulators. If so, it means all
mobile robots have reached their next waypoints or their goal. So break this loop
                                if all([item[0] == "Manipulator" for item in waiting_list]) and len(queue) == 0: # If all
are manipulators but no queue left to process
                                        print("\nCurrently all items in waiting list are Manipulators waiting to process,
and there are no mobile robots in queue. Continuing inner while loop.")
                                        continue
                                                                                                        # Continue without
exiting loop until waypoints exhaust

                                elif all([item[0] == "Manipulator" for item in waiting_list]) and len(queue) > 0:     # If
all are manipulators and there is a queue left to process
                                        print("\nCurrently all items in waiting list are Manipulators waiting to process,
but there are mobile robots in queue. Breaking inner while loop.")
                                        break

                                elif len(waiting_list) == 0 and len(queue) > 0:
                                        print("\nThere are mobile robots in the queue that have not reached their targets.
Computing their next waypoint. Breaking inner while loop.")
                                        break

                        queue = self.tasklist.getQueue()
                        if len(waiting_list) == 0 and len(queue) == 0:
                                print("\nNothing left in waiting list or in queue. Breaking outer while loop.")
                                break

                        elif len(waiting_list) == 0 and len(queue) > 0:
                                continue

                print("\nBoundless Mode Motion Complete! Congratulations!")

        def compute_next_waypoint(self, tuple_of_queue):

                print("\n\n\n===Start of compute_next_waypoint...")
```

```
#=============================================================================================================
          # 1) Make a robotlessmap
          currentmap = np.copy(self.information.grid)
          robotlessmap = np.copy(self.information.grid)
          timestampmap = np.copy(self.information.grid)

          for (robot, target, dtype, targetpose) in tuple_of_queue:
                    robotlessmap = self.removeRobotGridsFromMap(robotlessmap, self.objects["Mobile"][robot]["Grid"])

                    # Remove Debris from the map as well.
                    if dtype == "Debris":
                              robotlessmap = self.removeRobotGridsFromMap(robotlessmap,
self.objects["Debris"][target]["Grid"])



#=============================================================================================================
          # 2) Throw particles on the map
          z = self.objects["Mobile"].values()[0]["Seed"][0][-1]
          particles = set()
          for i in range(self.information.gx*self.information.gy*self.information.gz//(self.threshold*30)):
                    x = random.randint(0,self.information.gx-1)
                    y = random.randint(0,self.information.gy-1)
                    particles.add((x,y))



#=============================================================================================================
          # 3) Parse the Queue and see if any robot is absent, so we can discard them from self.waypoints
          if(len(self.waypoints) > 0):
                    temp = [item[0] for item in tuple_of_queue]
                    rk = self.waypoints.keys()
                    for robot in rk:
                              if robot not in temp:
                                        print(robot, "not in queue, so removing from self.waypoints")
                                        self.waypoints.pop(robot)



#=============================================================================================================
          # 4) Parse the Queue and initialize self.waypoints for each robot, for the first time.

          storage = {}
          for (robot, target, dtype, targetpose) in tuple_of_queue:

                    if robot not in self.waypoints.keys():
                              print("\n",robot," not in self.waypoints. Initializing it.")

                              self.waypoints[robot] = { \
                                        "Visited" : [], \
                                        "Agenda" : CostQueue(), \
                                        "CurrentWaypoints" : [], \
                                        "CurrentPoses" : [], \
                                        "Optimum" : None, \
                                        "NextPose" : None, \
                                        "InitPose" : self.objects["Mobile"][robot]["Pose"], \
                                        "InitGrid" : self.objects["Mobile"][robot]["Seed"][0], \
                                        "GoalGrid" : self.objects[dtype][target]["Seed"][0], \
                                        "GoalGrids" : self.objects[dtype][target]["Grid"], \
                                        "Iterations" : 0, \
                                        "Target" : target, \
                                        "Type" : dtype, \
                                        "TargetPose" : targetpose, \
                                        "Status" : False
                              }

                              self.recent[robot] = {    \
                                        "Waypoints" : None, \
                                        "Poses" : None, \
                                        "Collisions" : None
                              }

                              storage[robot] = None

                    else:
```

```
                        self.waypoints[robot]["Visited"] = []
                        self.waypoints[robot]["Agenda"] = CostQueue()
                        self.waypoints[robot]["NextPose"] = None
                        self.waypoints[robot]["InitPose"] = self.objects["Mobile"][robot]["Pose"]
                        self.waypoints[robot]["InitGrid"] = self.objects["Mobile"][robot]["Seed"][0]
                        self.waypoints[robot]["GoalGrid"] = self.objects[dtype][target]["Seed"][0]
                        self.waypoints[robot]["GoalGrids"] = self.objects[dtype][target]["Grid"]
                        self.waypoints[robot]["Iterations"] = 0
                        self.waypoints[robot]["Target"] = target
                        self.waypoints[robot]["Type"] = dtype
                        self.waypoints[robot]["TargetPose"] = targetpose
                        self.waypoints[robot]["Optimum"] = None
                        self.waypoints[robot]["Status"] = False
                        self.waypoints[robot]["CurrentWaypoints"] = []
                        self.waypoints[robot]["CurrentPoses"] = []

                        storage[robot] = None

                ROBOT_S_CURRENT_WAYPOINT =     {"CurrentSeed" : self.objects["Mobile"][robot]["Seed"][0], \
                                                                                "CurrentPose" :
self.objects["Mobile"][robot]["Pose"], \
                                                                                "CurrentMask" :
self.objects["Mobile"][robot]["Mask"], \
                                                                                "CurrentGrids" :
self.objects["Mobile"][robot]["Grid"], \
                                                                                "Collisions" : False }
                COST = distance_heuristic( self.waypoints[robot]["InitGrid"], self.waypoints[robot]["GoalGrid"])
                self.waypoints[robot]["Agenda"].push((COST, [ROBOT_S_CURRENT_WAYPOINT]))

                goal = self.waypoints[robot]["GoalGrid"][0:2]
                init = self.waypoints[robot]["InitGrid"][0:2]
                print("Robot ",robot," must reach from ",init," to ",goal)

                particles.add(goal)
                particles.add(init)


#=============================================================================================================
        # 5) Now make a graph using Delaunay Triangulation
        tparticles = [list(element) for element in particles]

        parts = [(p[0], p[1], z) for p in tparticles]
                publish_particles(parts, self.information, self.particles_pub, namespace="particles",clr=(0,1,0,0.3))


        tparticles = np.array(tparticles)
        tri = Delaunay(tparticles)
        graph = make_graph(tri)
        print("Triangulation Complete.\n\n")



#=============================================================================================================
        # 6) Now start the while loop

        MAX_ITER = 500
        while True:

                # Make a blank sketch map
                timestampmap = np.copy(robotlessmap)

                # Use a for loop to do some initializations for this iteration
                for (robot, target, dtype, targetpose) in tuple_of_queue:

                        # Check if Agenda is empty
                        if self.waypoints[robot]["Agenda"].isEmpty():
                                print("Agenda is empty for ",robot," so setting flag to True and quitting.")
                                print("This happens when robot cannot compute a path to the goal.")
                                self.waypoints[robot]["Status"] = True
                                continue

                        else:

                                if self.waypoints[robot]["Status"] is False:

                                        temp = self.waypoints[robot]["Agenda"].popQueue()
                                        self.waypoints[robot]["Optimum"] = copy.deepcopy(temp)
```

```
                                                #print("We are in initialization for loop. Optimum Agenda for ", robot, "
is ", [self.easypose(item["CurrentPose"]) for item in self.waypoints[robot]["Optimum"][1]])

                                                if self.waypoints[robot]["Optimum"][1][-1]["CurrentSeed"][0:2] ==
self.waypoints[robot]["GoalGrid"][0:2]:
                                                        print(robot, " has waypoints computed up to its goal.")
                                                        self.waypoints[robot]["Status"] = True

                                                        temp = self.waypoints[robot]["Optimum"]
                                                        storage[robot] = copy.deepcopy(temp)

                                                self.waypoints[robot]["Iterations"] += 1
                                                continue

                                        else:
                                                pass

                                # Put the robots on the timestampmap
                                timestampmap = self.addRobotGridsToMap(timestampmap, self.waypoints[robot]["Optimum"][1][-
1]["CurrentGrids"])

                        # Check if all robots have reached the goal
                        if all([self.waypoints[robot]["Status"] == True for robot in self.waypoints.keys()]):
                                print("All robots have paths computed to their goals. Breaking while loop.")
                                iterations = [(robotname, self.waypoints[robotname]["Iterations"]) for robotname in
self.waypoints.keys()]
                                print("Number of iterations it took for each robot are : ", iterations)
                                break


                        if any([self.waypoints[robot]["Iterations"] >= MAX_ITER for robot in self.waypoints.keys()]):
                                print("Too many iterations. Breaking while loop.")
                                iterations = [(robotname, self.waypoints[robotname]["Iterations"]) for robotname in
self.waypoints.keys()]
                                print("Number of iterations it took for each robot are : ", iterations)
                                break

                        tempmap = timestampmap

                        # Now we have done the initializations. We will move on to checking neighbouring particles.
                        for (robot, target, dtype, targetpose) in tuple_of_queue:

                                if self.waypoints[robot]["Status"] is False:

                                        tempmap = self.removeRobotGridsFromMap(tempmap,
self.waypoints[robot]["Optimum"][1][-1]["CurrentGrids"])

                                        initgrid = self.waypoints[robot]["Optimum"][1][-1]["CurrentSeed"]
                                        initpose = self.waypoints[robot]["Optimum"][1][-1]["CurrentPose"]
                                        initmask = self.waypoints[robot]["Optimum"][1][-1]["CurrentMask"]
                                        initgrids = self.waypoints[robot]["Optimum"][1][-1]["CurrentGrids"]
                                        goalgrid = self.waypoints[robot]["GoalGrid"]

                                        neighbours = get_neighbours(initgrid, tri, graph)

                                        rotlists = []

                                        for p in neighbours:

                                                if p in self.waypoints[robot]["Visited"]:
                                                        pass
                                                else:
                                                        t_initpose = copy.deepcopy(initpose)
                                                        rotlist = []
                                                        heuristic_cost = distance_heuristic(p, goalgrid)
                                                        translation = (p[0] - initgrid[0], p[1] - initgrid[1], 0)

                                                        for angle in [0, 90, -90]:

                                                                newmask, newpose, newgrids, rotcost, transcost =
self.get_transformations(robot, \


            t_initpose, \


            initgrids, \
```

```
            translation, \

            timestampmap, \

            endangle = angle, \

            rotincrement = 10, \

            transincrement = 2)
                                                                vlessmask, pose, vlessgrids =
self.get_transformed_mask_and_pose_only(robot, \
                                    t_initpose, \
                                    initgrids, \
                                    translation, \
                                    timestampmap, \
                                    endangle = angle)
                                                                collidingcost = collision_heuristic(newmask,
timestampmap)
                                                                #print("NewPose is ", self.easypose(newpose), " and
Pose is ", self.easypose(pose))
                                                                rotlist.append([collidingcost + rotcost +
heuristic_cost, p, newmask, newpose, newgrids, vlessmask, vlessgrids, collidingcost])

                                                        rotlist = sorted(rotlist, key = lambda x : x[0])
                                                        element = rotlist.pop(0)
                                                        rotlists.append(element)

                                    OList = copy.deepcopy(self.waypoints[robot]["Optimum"][1])
                                    #print("OList after deepcopy is : ", [self.easypose(item["CurrentPose"]) for item
in OList])

                                    for p in rotlists:
                                             OptimumList = None
                                             OptimumList = OList[:]

                                             COST = p[0]
                                             ROBOT_S_CURRENT_WAYPOINT =     {"CurrentSeed" : p[1], \

"CurrentPose" : p[3], \

"CurrentMask" : p[5], \

"CurrentGrids" : p[6], \

"Collisions" : False }
                                             if p[7] >= self.information.gx*self.information.gy*self.information.gz :
                                                      ROBOT_S_CURRENT_WAYPOINT["Collisions"] = True
                                             OptimumList.append(ROBOT_S_CURRENT_WAYPOINT)
                                             tempmap = self.addRobotGridsToMap(tempmap, p[4])
                                             #print("Appending OptimumList with particle ",
[self.easypose(item["CurrentPose"]) for item in OptimumList])

                                             self.waypoints[robot]["Agenda"].push((COST, OptimumList))
                                             self.waypoints[robot]["Visited"].append(p[1])

                                else:
                                    pass
                        #print("End of for loop.\n")


#=============================================================================================================
          # 7) Now we have computed all waypoints for this iteration. THERE WILL BE COLLISIONS. So now we have to do
collision checking and scheduling.
          #      Note that collision-checking and scheduling is specifically for the latest step. So we can have one
robot move to its next position while
```

```
          #      all others remain stationary. Due to dynamic replanning, once a robot reaches its goal and stays in the
way of another, the robot that
          #     gets 'stuck' will eventually find a way out.



          for (robot, target, dtype, targetpose) in tuple_of_queue:

                    colour = self.colors["Mobile"][robot]

                    if self.recent[robot]["Waypoints"] is None: # Waypoints, Poses, Collisions

                              new_wp = [wp["CurrentSeed"] for wp in storage[robot][1]] # The first element in this list is
where we are currently
                              new_collisions = [wp["Collisions"] for wp in storage[robot][1]] # The first element in this
list is where we are currently
                              new_poses = [wp["CurrentPose"] for wp in storage[robot][1]]

                              print("For ", robot, "we have:")
                              print("New Waypoints : ", new_wp)
                              print("New Collisions : ", new_collisions)
                              print("New Poses : ", [self.easypose(pose) for pose in new_poses])

                              if new_collisions[1] is False:
                                   print(robot," is not colliding")
                                        self.waypoints[robot]["NextPose"] = new_poses[1]

                                        self.recent[robot]["Waypoints"] = new_wp[:]
                                        self.waypoints[robot]["CurrentWaypoints"] = new_wp[:]

                                        self.recent[robot]["Poses"] = new_poses[:]
                                        self.recent[robot]["Collisions"] = new_collisions[:]
                                        publish_particles(new_wp, self.information, self.particles_pub,
namespace=robot+"/waypoints", clr=colour)
                                        print("\n",robot," : ", new_wp)
                                        print("\n", robot, " : ", [self.easypose(pose) for pose in
self.recent[robot]["Poses"]])

                              else:
                                   print(robot," is colliding, so we stop it.")
                                        self.waypoints[robot]["NextPose"] = new_poses[0]

                                        self.recent[robot]["Waypoints"] = new_wp[:]
                                        self.waypoints[robot]["CurrentWaypoints"] = new_wp[:]

                                        self.recent[robot]["Poses"] = new_poses[:]
                                        self.recent[robot]["Collisions"] = new_collisions[:]
                                        publish_particles(new_wp, self.information, self.particles_pub,
namespace=robot+"/waypoints", clr=colour)
                                        print("\n",robot," : ", new_wp)
                                        print("\n", robot, " : ", [self.easypose(pose) for pose in
self.recent[robot]["Poses"]])

                    else:

                              # For the 1st POSE we are going to subtract 90 degrees because this value keeps getting
erratically modified (temporary bugfix).


                              new_pathsum, old_pathsum = 0,0

                              new_wp = [wp["CurrentSeed"] for wp in storage[robot][1]] # The first element in this list is
where we are currently
                              new_collisions = [wp["Collisions"] for wp in storage[robot][1]] # The first element in this
list is where we are currently
                              new_poses = [wp["CurrentPose"] for wp in storage[robot][1]]

                              print("For ", robot, "we have:")
                              print("New Waypoints : ", new_wp)
                              print("New Collisions : ", new_collisions)
                              print("New Poses : ", [self.easypose(pose) for pose in new_poses])

                              old_wp = self.recent[robot]["Waypoints"][:]
                              old_poses = self.recent[robot]["Poses"][:]
                              old_collisions = self.recent[robot]["Collisions"][:]

                              faulty_pose = old_poses[1] # This will be a Pose() object
```

```
                            (rx, ry, rz, rw) = (faulty_pose.orientation.x,
faulty_pose.orientation.y,faulty_pose.orientation.z,faulty_pose.orientation.w)
                            (r,p,y) = euler_from_quaternion((rx, ry, rz, rw))
                            y -= math.radians(90)
                            quat = quaternion_from_euler(r,p,y)
                            faulty_pose.orientation.x = quat[0]
                            faulty_pose.orientation.y = quat[1]
                            faulty_pose.orientation.z = quat[2]
                            faulty_pose.orientation.w = quat[3]
                            old_poses[1] = faulty_pose


                            print("For ", robot, "we have:")
                            print("Old Waypoints : ", old_wp)
                            print("Old Collisions : ", old_collisions)
                            print("Old Poses : ", [self.easypose(pose) for pose in old_poses])


                            for i in range(1,len(new_wp)):
                                     new_pathsum += distance_heuristic(new_wp[i], new_wp[i-1])


                            for i in range(1,len(old_wp)):
                                     old_pathsum += distance_heuristic(old_wp[i], old_wp[i-1])

                            print("Old Path Sum : ", old_pathsum, ", New Path Sum : ", new_pathsum)

                            if new_pathsum < old_pathsum and new_collisions[1] is False:

                                     print("New pathsum is optimal and ",robot," is not colliding.")
                                     self.waypoints[robot]["NextPose"] = new_poses[1]
                                     self.recent[robot]["Waypoints"] = new_wp[:]
                                     self.waypoints[robot]["CurrentWaypoints"] = new_wp[:]

                                     self.recent[robot]["Poses"] = new_poses[:]
                                     self.recent[robot]["Collisions"] = new_collisions[:]
                                     publish_particles(new_wp[:], self.information, self.particles_pub,
namespace=robot+"/waypoints", clr=colour)
                                     print("\n",robot," : ", new_wp)
                                     print("\n", robot, " : ", [self.easypose(pose) for pose in
self.recent[robot]["Poses"]])


                            elif new_pathsum < old_pathsum and new_collisions[1] is True: # and old_collisions[1] is
True:

                                     print("New pathsum is optimal for ",robot," but it is colliding in new waypoints so
we stop the robot.")
                                     self.waypoints[robot]["NextPose"] = new_poses[0]

                                     self.recent[robot]["Waypoints"] = new_wp[:]
                                     self.waypoints[robot]["CurrentWaypoints"] = new_wp[:]

                                     self.recent[robot]["Collisions"] = new_collisions[:]
                                     self.recent[robot]["Poses"] = new_poses[:]
                                     publish_particles(new_wp[:], self.information, self.particles_pub,
namespace=robot+"/waypoints", clr=colour)
                                     print("\n",robot," : ", new_wp)
                                     print("\n", robot, " : ", [self.easypose(pose) for pose in
self.recent[robot]["Poses"]])

                            elif old_pathsum <= new_pathsum:

                                     print("Old pathsum is optimal for ",robot)

                                     # Check if the difference is very small
                                     nextpose = old_wp[1]
                                     currentpose = old_wp[0]
                                     if abs(currentpose[0] - nextpose[0]) <= 3 and abs(currentpose[1] - nextpose[1]) <=
3 and len(old_wp) > 1:
                                              old_wp.pop(1)
                                              old_poses.pop(1)
                                              old_collisions.pop(1)
                                 if old_collisions[1] is False:
                                     print("and it is not colliding.")
```

```
                                        self.waypoints[robot]["NextPose"] = old_poses[1]
                                        self.recent[robot]["Waypoints"] = old_wp[:]
                                        self.waypoints[robot]["CurrentWaypoints"] = old_wp[:]

                                        self.recent[robot]["Poses"] = old_poses[:]
                                        self.recent[robot]["Collisions"] = old_collisions[:]
                                        publish_particles(old_wp[:], self.information, self.particles_pub,
namespace=robot+"/waypoints", clr=colour)
                                        print("\n",robot," : ", old_wp)
                                        print("\n", robot, " : ", [self.easypose(pose) for pose in
self.recent[robot]["Poses"]])

                                else:
                                        if new_collisions[1] is False:
                                                print("but new path isn't colliding.")
                                                self.waypoints[robot]["NextPose"] = new_poses[1]
                                                self.recent[robot]["Waypoints"] = new_wp[:]
                                                self.waypoints[robot]["CurrentWaypoints"] = new_wp[:]

                                                self.recent[robot]["Collisions"] = new_collisions[:]
                                                self.recent[robot]["Poses"] = new_poses[:]
                                                publish_particles(new_wp[:], self.information,
self.particles_pub, namespace=robot+"/waypoints", clr=colour)
                                                print("\n",robot," : ", new_wp)
                                                print("\n", robot, " : ", [self.easypose(pose) for pose in
self.recent[robot]["Poses"]])
                                        else:
                                                print("and new path is also colliding. So we stay at the same
spot.")
                                                self.waypoints[robot]["NextPose"] = new_poses[0]
                                                self.recent[robot]["Waypoints"] = new_wp[:]
                                                self.waypoints[robot]["CurrentWaypoints"] = new_wp[:]

                                                self.recent[robot]["Poses"] = new_poses[:]
                                                self.recent[robot]["Collisions"] = new_collisions[:]
                                                publish_particles(new_wp[:], self.information,
self.particles_pub, namespace=robot+"/waypoints", clr=colour)
                                                print("\n",robot," : ", new_wp)
                                                print("\n", robot, " : ", [self.easypose(pose) for pose in
self.recent[robot]["Poses"]])

                            else:
                                    print("Collisions are expected for ",robot," so we stop the robot.")
                                    self.waypoints[robot]["NextPose"] = new_poses[0]
                                    self.recent[robot]["Waypoints"] = new_wp[:]
                                    self.waypoints[robot]["CurrentWaypoints"] = new_wp[:]

                                    self.recent[robot]["Poses"] = new_poses[:]
                                    self.recent[robot]["Collisions"] = new_collisions[:]
                                    publish_particles(new_wp[:], self.information, self.particles_pub,
namespace=robot+"/waypoints", clr=colour)
                                    print("\n",robot," : ", new_wp)
                                    print("\n", robot, " : ", [self.easypose(pose) for pose in
self.recent[robot]["Poses"]])



#=============================================================================================================
          # And that's it! You have computed the waypoints!

          print("End of compute_next_waypoint.")

          # 7) End of compute. Do some Wrap-Ups.

    def easypose(self, end_eff_pose):
          if isinstance(end_eff_pose, Pose):
                    origin = (end_eff_pose.position.x, end_eff_pose.position.y, end_eff_pose.position.z, 0) # These are
coordinates in world frame. This is a quaternion-like tuple
                    q2 = (end_eff_pose.orientation.x, end_eff_pose.orientation.y, end_eff_pose.orientation.z,
end_eff_pose.orientation.w)

          elif isinstance(end_eff_pose, TransformStamped):
                    origin = (end_eff_pose.transform.translation.x, end_eff_pose.transform.translation.y,
end_eff_pose.transform.translation.z, 0) # These are coordinates in world frame. This is a quaternion-like tuple
                    q2 = (end_eff_pose.transform.rotation.x, end_eff_pose.transform.rotation.y,
end_eff_pose.transform.rotation.z, end_eff_pose.transform.rotation.w)
```

```
          (r,p,y) = euler_from_quaternion((q2[0], q2[1], q2[2], q2[3]))

          return (round(math.degrees(y),5))

    def removeRobotGridsFromMap(self, timestampmap, grids):
          for index in grids:
                    if index[0] >=0 and index[0] < self.information.gx and index[1] >=0 and index[1] <
self.information.gy and index[2] >=0 and index[2] < self.information.gz:
                              timestampmap[index[0]][index[1]][index[2]] = 0
          return timestampmap

    def addRobotGridsToMap(self, timestampmap, grids):
          for index in grids:
                    if index[0] >=0 and index[0] < self.information.gx and index[1] >=0 and index[1] <
self.information.gy and index[2] >=0 and index[2] < self.information.gz:
                              timestampmap[index[0]][index[1]][index[2]] = 1
          return timestampmap

    def get_transformations(self, robotname, end_eff_pose, current_cells, \
          translation = (0,0,0), initmap = None, endangle = 0, rotincrement=10, transincrement=2, robot_type="Mobile"):
          """This function must return : Mask, Pose, RotCost, TransCost, Grids"""

          if isinstance(end_eff_pose, Pose):
                    origin = (end_eff_pose.position.x, end_eff_pose.position.y, end_eff_pose.position.z, 0) # These are
coordinates in world frame. This is a quaternion-like tuple
                    q2 = (end_eff_pose.orientation.x, end_eff_pose.orientation.y, end_eff_pose.orientation.z,
end_eff_pose.orientation.w)

          elif isinstance(end_eff_pose, TransformStamped):
                    origin = (end_eff_pose.transform.translation.x, end_eff_pose.transform.translation.y,
end_eff_pose.transform.translation.z, 0) # These are coordinates in world frame. This is a quaternion-like tuple
                    q2 = (end_eff_pose.transform.rotation.x, end_eff_pose.transform.rotation.y,
end_eff_pose.transform.rotation.z, end_eff_pose.transform.rotation.w)

          (r,p,y) = euler_from_quaternion((q2[0], q2[1], q2[2], q2[3]))
          startangle = 0

          points = convert_indices_list_to_quat_coordinates(current_cells, self.information) # Returns the grid indices
as quat-like tuples of coordinates

          set_of_points = set()
          vector_of_transformed_points = []

          # Subtract cloudpoint-wrt-world - originpoint-wrt-world to get vectors P1
          for pt in points:
                    set_of_points.add((pt[0]-origin[0], pt[1]-origin[1], pt[2]-origin[2], 0))



          # q2 is old quaternion, q1 is new quaternion
          if startangle >= 0 and endangle >=  0:
                    for i in range(startangle, endangle + rotincrement, rotincrement):
                              q1 = angle_correction(q2,i)

                              q2_inverse = inverse4(q2)
                              qdiff = vector_multiply4(q1, q2_inverse)
                              qdiffn = normalize(qdiff)

                              for point in set_of_points:
                                        pt = vector_multiply4(vector_multiply4(qdiffn, point), inverse4(qdiffn))
                                        pt = (pt[0] + origin[0], pt[1] + origin[1], pt[2] + origin[2])
                                        vector_of_transformed_points.append(pt)
          elif startangle >= 0 and endangle < 0:
                    for i in range(startangle, endangle - rotincrement, -rotincrement):
                              q1 = angle_correction(q2,i)

                              q2_inverse = inverse4(q2)
                              qdiff = vector_multiply4(q1, q2_inverse)
                              qdiffn = normalize(qdiff)

                              for point in set_of_points:
                                        pt = vector_multiply4(vector_multiply4(qdiffn, point), inverse4(qdiffn))
                                        pt = (pt[0] + origin[0], pt[1] + origin[1], pt[2] + origin[2])
                                        vector_of_transformed_points.append(pt)
          elif startangle < 0 and endangle >= 0: # Such as the case of -90 to +180 or -90 to +90
                    for i in range(startangle, endangle + rotincrement, rotincrement):
                              q1 = angle_correction(q2,i)
```

```
                        q2_inverse = inverse4(q2)
                        qdiff = vector_multiply4(q1, q2_inverse)
                        qdiffn = normalize(qdiff)

                        for point in set_of_points:
                                pt = vector_multiply4(vector_multiply4(qdiffn, point), inverse4(qdiffn))
                                pt = (pt[0] + origin[0], pt[1] + origin[1], pt[2] + origin[2])
                                vector_of_transformed_points.append(pt)
        elif startangle < 0 and endangle < 0:
                for i in range(startangle, endangle - rotincrement, -rotincrement):
                        q1 = angle_correction(q2,i)

                        q2_inverse = inverse4(q2)
                        qdiff = vector_multiply4(q1, q2_inverse)
                        qdiffn = normalize(qdiff)

                        for point in set_of_points:
                                pt = vector_multiply4(vector_multiply4(qdiffn, point), inverse4(qdiffn))
                                pt = (pt[0] + origin[0], pt[1] + origin[1], pt[2] + origin[2])
                                vector_of_transformed_points.append(pt)




        gidxs = find_grid_idx(vector_of_transformed_points, self.information)
        tempset = set()
        for item in gidxs:
                tempset.add(item)
        gidxs = list(tempset)
        tempgidxs = gidxs

        startgrid = list(self.objects[robot_type][robotname]["Seed"][0])
        endgrid = [startgrid[0] + translation[0], startgrid[1] + translation[1],startgrid[2] + translation[2]]
        translations = [[0,0]]
        currentpos = startgrid
        while True:
                currentpos = [currentpos[0] + translations[-1][0], currentpos[1] + translations[-1][1], currentpos[2]
+ 0]
                if currentpos == endgrid:
                        break
                newpossouth = [currentpos[0] + 1, currentpos[1], currentpos[2]]
                costsouth = math.sqrt((endgrid[0]-newpossouth[0])**2 + (endgrid[1]-newpossouth[1])**2)
                newposnorth = [currentpos[0] - 1, currentpos[1], currentpos[2]]
                costnorth = math.sqrt((endgrid[0]-newposnorth[0])**2 + (endgrid[1]-newposnorth[1])**2)
                newposeast = [currentpos[0], currentpos[1]+1, currentpos[2]]
                costeast = math.sqrt((endgrid[0]-newposeast[0])**2 + (endgrid[1]-newposeast[1])**2)
                newposwest = [currentpos[0], currentpos[1]-1, currentpos[2]]
                costwest = math.sqrt((endgrid[0]-newposwest[0])**2 + (endgrid[1]-newposwest[1])**2)
                queue = [((1,0),costsouth), ((-1,0),costnorth), ((0,1),costeast), ((0,-1),costwest)]
                queue = sorted(queue, key = lambda x : x[1])
                elem = queue.pop(0)
                translations.append(list(elem[0]))

        mask = np.zeros_like(self.information.grid)
        for trans in translations:
                tempgidxs = translate_grid_idx(tempgidxs, (trans[0],trans[1],0))
                mask = np.logical_or(mask, make_binary_mask(tempgidxs, initmap)).astype(int)

        ogrids = self.get_indices_from_mask(mask)

        returnpose = Pose()
        rot = angle_correction(q2, endangle)

        trans = (origin[0] + translation[0]*self.information.resolution_x, \
                        origin[1] + translation[1]*self.information.resolution_y, \
                        origin[2] + translation[2]*self.information.resolution_z)
        returnpose.orientation.x = rot[0]
        returnpose.orientation.y = rot[1]
        returnpose.orientation.z = rot[2]
        returnpose.orientation.w = rot[3]
        returnpose.position.x = trans[0]
        returnpose.position.y = trans[1]
        returnpose.position.z = trans[2]

        #print("Transformation pose ", self.easypose(returnpose))
```

```
          return mask, returnpose, ogrids, abs(int((endangle - startangle)/rotincrement)), math.sqrt(translation[0]**2 +
translation[1]**2 + 0)

    def get_transformed_mask_and_pose_only(self, robotname, end_eff_pose, current_cells, \
          translation = (0,0,0), initmap = None, endangle = 0, robot_type="Mobile"):
          """This function must return : Mask, Pose, Grids"""

          if isinstance(end_eff_pose, Pose):
                    origin = (end_eff_pose.position.x, end_eff_pose.position.y, end_eff_pose.position.z, 0) # These are
coordinates in world frame. This is a quaternion-like tuple
                    q2 = (end_eff_pose.orientation.x, end_eff_pose.orientation.y, end_eff_pose.orientation.z,
end_eff_pose.orientation.w)

          elif isinstance(end_eff_pose, TransformStamped):
                    origin = (end_eff_pose.transform.translation.x, end_eff_pose.transform.translation.y,
end_eff_pose.transform.translation.z, 0) # These are coordinates in world frame. This is a quaternion-like tuple
                    q2 = (end_eff_pose.transform.rotation.x, end_eff_pose.transform.rotation.y,
end_eff_pose.transform.rotation.z, end_eff_pose.transform.rotation.w)

          (r,p,y) = euler_from_quaternion((q2[0], q2[1], q2[2], q2[3]))

          points = convert_indices_list_to_quat_coordinates(current_cells, self.information) # Returns the grid indices
as quat-like tuples of coordinates

          set_of_points = set()
          vector_of_transformed_points = []

          # Subtract cloudpoint-wrt-world - originpoint-wrt-world to get vectors P1
          for pt in points:
                    set_of_points.add((pt[0]-origin[0], pt[1]-origin[1], pt[2]-origin[2], 0))

          q1 = angle_correction(q2,endangle)

          q2_inverse = inverse4(q2)
          qdiff = vector_multiply4(q1, q2_inverse)
          qdiffn = normalize(qdiff)



          for point in set_of_points:
                    pt = vector_multiply4(vector_multiply4(qdiffn, point), inverse4(qdiffn))
                    pt = (pt[0] + origin[0], pt[1] + origin[1], pt[2] + origin[2])
                    vector_of_transformed_points.append(pt)



          gidxs = find_grid_idx(vector_of_transformed_points, self.information)
          tempset = set()
          for item in gidxs:
                    tempset.add(item)
          gidxs = list(tempset)



          tempgidxs = translate_grid_idx(gidxs, (translation[0],translation[1],0))
          mask = make_binary_mask(tempgidxs, initmap)

          ogrids = self.get_indices_from_mask(mask)

          returnpose = Pose()
          trans = (origin[0] + translation[0]*self.information.resolution_x, \
                             origin[1] + translation[1]*self.information.resolution_y, \
                             origin[2] + translation[2]*self.information.resolution_z)
          returnpose.orientation.x = q1[0]
          returnpose.orientation.y = q1[1]
          returnpose.orientation.z = q1[2]
          returnpose.orientation.w = q1[3]
          returnpose.position.x = trans[0]
          returnpose.position.y = trans[1]
          returnpose.position.z = trans[2]

          #print("Transformed Mask Only pose ", self.easypose(returnpose))

          return mask, returnpose, ogrids

    def get_indices_from_mask(self, mask):
          idxs = np.transpose(np.nonzero(mask))
          return [tuple(item) for item in idxs]
```

```
def get_random_particle(self, startgrid, threshold, rparticles, ptype_flag=False):
	z = self.objects["Mobile"].values()[0]["Seed"][0][-1]

	space = []
	for x in range(startgrid[0]-threshold, startgrid[0]+threshold):
		for y in range(startgrid[1]-threshold, startgrid[1] + threshold):
			if x >= 0 and x < self.information.gx and y >= 0 and y < self.information.gy:
				space.append((x,y,z))


	if ptype_flag is True:
		for particle in space:
			if (particle[0],particle[1]) in self.forbiddenzone:
				space.remove((particle[0], particle[1], z))


	for particle in space:
		if (particle[0], particle[1], z) in rparticles:
			space.remove((particle[0], particle[1], z))

	[sample] = random.sample(space,1)

	return sample



def update_velocities(self, feedback):
	"""Update the velocities as received from the robots"""
	linear_x = feedback.vx
	linear_y = feedback.vy
	linear_z = feedback.vz
	angular_x = feedback.ax
	angular_y = feedback.ay
	angular_z = feedback.az
	t = feedback.time
	robotname = feedback.robotname
	robottype = feedback.robottype
	self.grid_velocities[robottype][robotname] = (linear_x, linear_y, linear_z, angular_x, angular_y, angular_z)

def signal_target_deletion(self, targetframe, dtype):
	"""Send a signal to obstacle spawner to remove the target"""
	req = SpawnObstaclesRequest()
	req.name = targetframe
	req.type = dtype
	req.x = 0.0
	req.y = 0.0
	req.z = 0.0
	result = self.obstacle_deletion(req)
	print("\nSent Request for Target Deletion.")
	while True:
		if targetframe in self.objects[dtype].keys():
			print("Found ",targetframe," in dictionary. Waiting for it to be deleted...")
			self.objects[dtype].pop(targetframe)
			if len(self.objects[dtype]) == 0:
				self.objects.pop(dtype)
			rospy.sleep(0.1)
			break
		else:
			continue
	print("\nTarget deleted.")

def check_expected_collisions_in_manipulators(self, robotname):
	"""Check collisions in Manipulators"""

	RADIUS = 2

	# Distance-based implementation
	debrises = self.objects["Debris"].keys()
	mrobots = self.objects["Mobile"].keys()
	for dbr in debrises:
		debrispos = self.objects["Debris"][dbr]["Seed"][0]
		manippos = self.objects["Manipulator"][robotname]["Seed"][0]
		if distance_heuristic(debrispos, manippos) <= RADIUS:
			return True

	for mbr in mrobots:
		mbrpos = self.objects["Mobile"][mbr]["Seed"][0]
```

```
                    manippos = self.objects["Manipulator"][robotname]["Seed"][0]
                    if distance_heuristic(mbrpos, manippos) <= RADIUS:
                              return True

          return False



def main(args):
          rospy.init_node('commander_node', anonymous=True)
          cm = Commander()
          try:
                    rospy.spin()
          except KeyboardInterrupt:
                    print("Shutting down!")

if __name__ == '__main__':
    """Main Function"""
    main(sys.argv)
```

APPENDIX B

Note: The code provided here is part of a larger repository, and only reflects the task of immediate interest. For reference, the repository can be accessed at: https://github.com/ankurjay/ankurbot/tree/multi_processing_trials_skidsteer_astar

Relative paths are provided per this repository, for reference.

C++ Code for Filtering, Merging, Voxelizing and Clustering Point Clouds

Path: ankurjay/ankurbot/ankurbot_description/src/merge_clouds.cpp

```
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
#include <tf2_ros/message_filter.h>
#include <tf2_ros/transform_listener.h>
#include <tf2_sensor_msgs/tf2_sensor_msgs.h> // REQUIRES INSTALLATION OF TF2 SENSOR MSGS from APT
#include <message_filters/subscriber.h>

#include <boost/thread/mutex.hpp>
#include <boost/shared_ptr.hpp>
#include <boost/bind.hpp>

#include <algorithm>

#include <pcl_conversions/pcl_conversions.h>
#include <pcl/point_types.h>
#include <pcl_ros/point_cloud.h>
#include <pcl/point_cloud.h>

//PCL specific includes
#include <pcl/io/pcd_io.h> //Required for pcl::toROSMsg and pcl::romROSMsg
#include <pcl/conversions.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/filters/crop_box.h>
#include <pcl/search/search.h>
#include <pcl/search/kdtree.h>
#include <pcl/features/normal_3d.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/extract_clusters.h>

// Custom Message
#include "ankurbot_multisim/CloudVector.h"

/*
Because we have the Seed and Cluster classes, we don't need to make our own class. Simply use ankurbot_multisim::Seed and
ankurbot_multisim::Cluster
*/

/*using Seed = ankurbot_multisim::Seed;
using Cluster = ankurbot_multisim::Cluster;

// Need to overload these operators because we are using Sets
namespace ankurbot_multisim {
    bool operator<(const Seed& s1, const Seed& s2) {return s1.seed_name < s2.seed_name; }
    bool operator>(const Seed& s1, const Seed& s2) {return s1.seed_name > s2.seed_name; }
}
*/


class MergeClouds {
public:
    MergeClouds(ros::NodeHandle& private_nh_) :
```

```
    tf2_listener(buffer_),
    target_frame_("world")
    {
          cloud1_sub_.subscribe(nh_, "/mapping_camera_1_mapping_kinect/depth/points", 10); //This is a MessageFilter, not
a regular subscriber
          cloud2_sub_.subscribe(nh_, "/mapping_camera_2_mapping_kinect/depth/points", 10); //This is a MessageFilter, not
a regular subscriber
          cloud3_sub_.subscribe(nh_, "/mapping_camera_3_mapping_kinect/depth/points", 10); //This is a MessageFilter, not
a regular subscriber
          cloud4_sub_.subscribe(nh_, "/mapping_camera_4_mapping_kinect/depth/points", 10); //This is a MessageFilter, not
a regular subscriber
          voxelized_sub_.subscribe(nh_,"voxelized_cloud",10);

         tf2_message_filter1_.reset(new tf2_ros::MessageFilter<pcl::PointCloud<pcl::PointXYZRGB>>(cloud1_sub_, buffer_,
target_frame_, 10, 0));
          tf2_message_filter1_->registerCallback(boost::bind(&MergeClouds::callback1, this, _1));
         tf2_message_filter2_.reset(new tf2_ros::MessageFilter<pcl::PointCloud<pcl::PointXYZRGB>>(cloud2_sub_, buffer_,
target_frame_, 10, 0));
          tf2_message_filter2_->registerCallback(boost::bind(&MergeClouds::callback2, this, _1));

         tf2_message_filter3_.reset(new tf2_ros::MessageFilter<pcl::PointCloud<pcl::PointXYZRGB>>(cloud3_sub_, buffer_,
target_frame_, 10, 0));
          tf2_message_filter3_->registerCallback(boost::bind(&MergeClouds::callback3, this, _1));
         tf2_message_filter4_.reset(new tf2_ros::MessageFilter<pcl::PointCloud<pcl::PointXYZRGB>>(cloud4_sub_, buffer_,
target_frame_, 10, 0));
          tf2_message_filter4_->registerCallback(boost::bind(&MergeClouds::callback4, this, _1));

          voxelized_sub_.registerCallback(boost::bind(&MergeClouds::clusterCallback, this, _1));

         private_nh_.param<double>("max_frequency", max_freq_, 1000.0);
         private_nh_.param<double>("cluster_radius", radius, 0.3);

         private_nh_.param<float>("voxel_size_x", xleafsize, 0.1f);
         private_nh_.param<float>("voxel_size_y", yleafsize, 0.1f);
         private_nh_.param<float>("voxel_size_z", zleafsize, 0.1f);

         private_nh_.param<double>("xmin", xmin, -2.5);
         private_nh_.param<double>("xmax", xmax, 2.5);
         private_nh_.param<double>("ymin", ymin, -2.5);
         private_nh_.param<double>("ymax", ymax, 2.5);
         private_nh_.param<double>("zmin", zmin, 0.01);
         private_nh_.param<double>("zmax", zmax, 1.3);



          // This TIMER is absolutely required
                  // make sure we don't publish too fast
         if (max_freq_ > 1000.0 || max_freq_ < 0.0)
         max_freq_ = 0.0;

         if (max_freq_ > 0.0)
         {
         timer_ = nh_.createTimer(ros::Duration(1.0/max_freq_), boost::bind(&MergeClouds::onTimer, this, _1));
         haveTimer_ = true;
         }
         else
         haveTimer_ = false;



          voxelized_pub_ = nh_.advertise<sensor_msgs::PointCloud2>("voxelized_cloud", 5);
          clusters_pub_ = nh_.advertise<ankurbot_multisim::CloudVector>("cloud_clusters", 5);

         newCloud1_ = newCloud2_ = newCloud3_ = newCloud4_ = false;

/*        xmin = -2.5;
          xmax = 2.5;
          ymin = -2.5;
          ymax = 2.5;
          zmin = 0.01;
          zmax = 1.3;

          xleafsize = 0.1f;
          yleafsize = 0.1f;
          zleafsize = 0.1f;
*/    }

    void onTimer(const ros::TimerEvent &e)
    {
```

```
	if (newCloud1_ && newCloud2_ && newCloud3_ && newCloud4_ )
		MergeAndVoxelizeCloud();
    }



    void callback1(const pcl::PointCloud<pcl::PointXYZRGB>::ConstPtr& cloudptr) {
	lock1_.lock();
	pcl::PointCloud<pcl::PointXYZRGB>::Ptr temp (new pcl::PointCloud<pcl::PointXYZRGB>);
	transformCloud(cloudptr, *temp);
	limitCloud(temp, cloud1_); // However, this is serializing the process. So I need some other process that is
continuously asking for limiting and publishing
	lock1_.unlock();
	newCloud1_ = true;
    /*	if (!haveTimer_ && newCloud2_) ////////////////////////This is only if maxfreq is set to 0.
		publishClouds(); */
    /*
    merge_clouds_node: /usr/include/boost/smart_ptr/shared_ptr.hpp:728: typename boost::detail::sp_dereference<T>::type
boost::shared_ptr<T>::operator*() const [with T = pcl::PointCloud<pcl::PointXYZRGB>; typename
boost::detail::sp_dereference<T>::type = pcl::PointCloud<pcl::PointXYZRGB>&]: Assertion `px != 0' failed.
    This error means that we are trying to access a shared_ptr without initializing it with the new keyword.
    */
    }



    void callback2(const pcl::PointCloud<pcl::PointXYZRGB>::ConstPtr& cloudptr) {
	lock2_.lock();
	pcl::PointCloud<pcl::PointXYZRGB>::Ptr temp (new pcl::PointCloud<pcl::PointXYZRGB>);
	transformCloud(cloudptr, *temp);
	limitCloud(temp, cloud2_);
	lock2_.unlock();
	newCloud2_ = true;
    /*	if (!haveTimer_ && newCloud1_) ////////////////////////
		publishClouds(); */
    }


    void callback3(const pcl::PointCloud<pcl::PointXYZRGB>::ConstPtr& cloudptr) {
	lock3_.lock();
	pcl::PointCloud<pcl::PointXYZRGB>::Ptr temp (new pcl::PointCloud<pcl::PointXYZRGB>);
	transformCloud(cloudptr, *temp);
	limitCloud(temp, cloud3_);
	lock3_.unlock();
	newCloud3_ = true;
    /*	if (!haveTimer_ && newCloud1_) ////////////////////////
		publishClouds(); */
    }


    void callback4(const pcl::PointCloud<pcl::PointXYZRGB>::ConstPtr& cloudptr) {
	lock4_.lock();
	pcl::PointCloud<pcl::PointXYZRGB>::Ptr temp (new pcl::PointCloud<pcl::PointXYZRGB>);
	transformCloud(cloudptr, *temp);
	limitCloud(temp, cloud4_);
	lock4_.unlock();
	newCloud4_ = true;
    /*	if (!haveTimer_ && newCloud1_) ////////////////////////
		publishClouds(); */
    }


    void clusterCallback(const pcl::PointCloud<pcl::PointXYZRGB>::ConstPtr& cloudptr) {
	cluster_lock_.lock();

	ankurbot_multisim::CloudVector c;
	sensor_msgs::PointCloud2 temp;
	pcl::search::KdTree<pcl::PointXYZRGB>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZRGB>);
	tree->setInputCloud (cloudptr);
	std::vector<pcl::PointIndices> clusters;
	pcl::EuclideanClusterExtraction<pcl::PointXYZRGB> ec;
	ec.setClusterTolerance (0.2); //0.02 = 2cm
	ec.setMinClusterSize (5);
	ec.setMaxClusterSize (25000);
	ec.setSearchMethod (tree);
```

```
          ec.setInputCloud (cloudptr);
          ec.extract (clusters);

          for (std::vector<pcl::PointIndices>::const_iterator it = clusters.begin (); it != clusters.end (); ++it) {

                  pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_cluster (new pcl::PointCloud<pcl::PointXYZRGB>);

                  for (std::vector<int>::const_iterator pit = it->indices.begin (); pit != it->indices.end (); ++pit)
                            cloud_cluster->push_back ((*cloudptr)[*pit]);

                   cloud_cluster->width = cloud_cluster->size ();
                   cloud_cluster->height = 1;
                   cloud_cluster->is_dense = false;
                   cloud_cluster->header = cloudptr->header;
                   pcl::toROSMsg(*cloud_cluster, temp);
                   c.clouds.push_back(temp);

          }
          publishData<ankurbot_multisim::CloudVector>(c, clusters_pub_);
          cluster_lock_.unlock();
    }


    void transformCloud(const pcl::PointCloud<pcl::PointXYZRGB>::ConstPtr& cloud, pcl::PointCloud<pcl::PointXYZRGB>&
cloudOut) {

          //==========================Apply
Transformation=============================================================================

          if(target_frame_ != cloud->header.frame_id) {
                   geometry_msgs::TransformStamped transformStamped;
                   try {
                  transformStamped = buffer_.lookupTransform(target_frame_, cloud->header.frame_id,ros::Time::now(),
ros::Duration(3.0));
                  //This is a blocking operation until either the transform is found or the timeout is reached.

                   }
                   catch (tf2::TransformException& ex) {
                            ROS_WARN("Could NOT transform the clouds");
                   }
         //Input and Output should be of ROS type. So PointCloud2 type.
         sensor_msgs::PointCloud2 temp1, temp2;

         pcl::toROSMsg(*cloud, temp1);
         //temp1 now has all header information from filtered_cloudPtr. temp2 has nothing

         tf2::doTransform(temp1, temp2, transformStamped); //This transformation is possible due to tf2_sensor_msgs
package
         // It copies header information from transformStamped to temp2

         pcl::fromROSMsg(temp2, cloudOut);
          }
         else
                   cloudOut = *cloud;
    }

    void limitCloud(const pcl::PointCloud<pcl::PointXYZRGB>::ConstPtr& cloud, pcl::PointCloud<pcl::PointXYZRGB>&
cloudOut) {
          pcl::CropBox<pcl::PointXYZRGB> boxFilter;
          boxFilter.setMin(Eigen::Vector4f(xmin, ymin, zmin, 1.0));
          boxFilter.setMax(Eigen::Vector4f(xmax, ymax, zmax, 1.0));
          boxFilter.setInputCloud(cloud);
          boxFilter.filter(cloudOut);
    }

    void MergeAndVoxelizeCloud() {
          newCloud1_ = false;
          newCloud2_ = false;
          newCloud3_ = false;
          newCloud4_ = false;

          pcl::PointCloud<pcl::PointXYZRGB>::Ptr combined (new pcl::PointCloud<pcl::PointXYZRGB>);

          *combined += cloud1_;
          *combined += cloud2_;
          *combined += cloud3_;
          *combined += cloud4_;
```

```
          combined->header.stamp = std::max(cloud1_.header.stamp, std::max(cloud2_.header.stamp,
std::max(cloud3_.header.stamp, cloud4_.header.stamp)));
          if (cloud1_.header.stamp ==  combined->header.stamp)
                   combined->header = cloud1_.header;
          else if (cloud2_.header.stamp ==  combined->header.stamp)
                   combined->header = cloud2_.header;
          else if (cloud3_.header.stamp ==  combined->header.stamp)
                   combined->header = cloud3_.header;
          else if (cloud4_.header.stamp ==  combined->header.stamp)
                   combined->header = cloud4_.header;

          //==================================VOXELIZE=====================================================

          // combined is a pcl::PointCloud<pcl::PointXYZRGB>::Ptr
          pcl::PointCloud<pcl::PointXYZRGB>::Ptr voxelized (new pcl::PointCloud<pcl::PointXYZRGB>); // This is the output
cloud ptr

          pcl::VoxelGrid<pcl::PointXYZRGB> voxgrid;
          voxgrid.setInputCloud (combined); //This should be the combined cloud
          voxgrid.setLeafSize (xleafsize, yleafsize, zleafsize);
          voxgrid.filter (*voxelized);
          publishClouds(voxelized, voxelized_pub_);
    }


    void publishClouds(const pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud, ros::Publisher& pub) {
          sensor_msgs::PointCloud2 ros_cloud;
          pcl::toROSMsg(*cloud, ros_cloud);
          pub.publish(ros_cloud);
    }

    template<class T>
    void publishData(const T& data, ros::Publisher& pub) {
          pub.publish(data);
    }



private:
    std::string target_frame_;

    tf2_ros::Buffer buffer_;
    tf2_ros::TransformListener tf2_listener;

    ros::NodeHandle nh_;

    std::string initial_cloud;

    message_filters::Subscriber<pcl::PointCloud<pcl::PointXYZRGB>> cloud1_sub_;
    message_filters::Subscriber<pcl::PointCloud<pcl::PointXYZRGB>> cloud2_sub_;
    message_filters::Subscriber<pcl::PointCloud<pcl::PointXYZRGB>> cloud3_sub_;
    message_filters::Subscriber<pcl::PointCloud<pcl::PointXYZRGB>> cloud4_sub_;
    message_filters::Subscriber<pcl::PointCloud<pcl::PointXYZRGB>> voxelized_sub_;


    boost::shared_ptr<tf2_ros::MessageFilter<pcl::PointCloud<pcl::PointXYZRGB>> > tf2_message_filter1_;
    boost::shared_ptr<tf2_ros::MessageFilter<pcl::PointCloud<pcl::PointXYZRGB>> > tf2_message_filter2_;
    boost::shared_ptr<tf2_ros::MessageFilter<pcl::PointCloud<pcl::PointXYZRGB>> > tf2_message_filter3_;
    boost::shared_ptr<tf2_ros::MessageFilter<pcl::PointCloud<pcl::PointXYZRGB>> > tf2_message_filter4_;



    pcl::PointCloud<pcl::PointXYZRGB> cloud1_;
    pcl::PointCloud<pcl::PointXYZRGB> cloud2_;
    pcl::PointCloud<pcl::PointXYZRGB> cloud3_;
    pcl::PointCloud<pcl::PointXYZRGB> cloud4_;
    pcl::PointCloud<pcl::PointXYZRGB> limited_cloud_;



    ros::Timer timer_;
    bool haveTimer_;
    double max_freq_;
    double radius;                                                  //Radius for searching in cluster

    bool newCloud1_;
    bool newCloud2_;
```

```
    bool newCloud3_;
    bool newCloud4_;

    boost::mutex lock1_; // A mutex prevents other threads from taking ownership while a particular thread owns the mutex
    boost::mutex lock2_;
    boost::mutex lock3_;
    boost::mutex lock4_;
    boost::mutex cluster_lock_;

    ros::Publisher voxelized_pub_;
    ros::Publisher clusters_pub_;

    double xmax;
    double xmin;
    double ymax;
    double ymin;
    double zmax;
    double zmin;

    float xleafsize;
    float yleafsize;
    float zleafsize;
};

int main(int argc, char** argv)
{
    ros::init(argc, argv, "merge_clouds_node");
    ros::NodeHandle private_nh_("~");
    MergeClouds mc(private_nh_);
    ros::spin();
    return 0;
}
```

C++ Code for Cluster Association

Path: ankurjay/ankurbot/ankurbot_description/src/cluster_assoc.cpp

```
// Headers
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
#include <tf2_ros/message_filter.h>
#include <tf2_ros/transform_listener.h>
#include <tf2_sensor_msgs/tf2_sensor_msgs.h> // REQUIRES INSTALLATION OF TF2 SENSOR MSGS from APT
#include <message_filters/subscriber.h>
#include <std_msgs/Int8.h>
#include <boost/thread/mutex.hpp>
#include <boost/shared_ptr.hpp>
#include <boost/bind.hpp>

#include <algorithm>

#include <pcl_conversions/pcl_conversions.h>
#include <pcl/point_types.h>
#include <pcl_ros/point_cloud.h>
#include <pcl/point_cloud.h>

//PCL specific includes
#include <pcl/io/pcd_io.h> //Required for pcl::toROSMsg and pcl::romROSMsg
#include <pcl/conversions.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/filters/crop_box.h>
#include <pcl/search/search.h>
#include <pcl/search/kdtree.h>
#include <pcl/kdtree/kdtree_flann.h>

#include <geometry_msgs/TransformStamped.h>
#include <vector>
#include <string>
```

```
#include <unordered_map>


// Custom Message
#include "ankurbot_multisim/CloudVector.h"
#include "ankurbot_multisim/TransformStampedVector.h"
#include "ankurbot_multisim/Cluster.h"
#include "ankurbot_multisim/AssociatedCluster.h"
#include "ankurbot_multisim/AssociatedClusterVector.h"
#include "ankurbot_multisim/CloudPoints.h"

#include "transforms.h"

using Seeds = ankurbot_multisim::TransformStampedVector;
using Clouds = ankurbot_multisim::CloudVector;
using Clusters = ankurbot_multisim::Cluster;
using ACluster = ankurbot_multisim::AssociatedCluster;
using ACVector = ankurbot_multisim::AssociatedClusterVector;
using cpoints = ankurbot_multisim::CloudPoints;



// Structs - Seed and CloudProperties
struct Seed {
    geometry_msgs::TransformStamped initial_transform;
    geometry_msgs::TransformStamped recent_transform;
    geometry_msgs::TransformStamped new_transform;
    std::string label;
    std::string frame;
    bool flag;
    std::string object;
    pcl::PointCloud<pcl::PointXYZRGB> subtraction_cloud;
    pcl::PointCloud<pcl::PointXYZRGB> initial_cloud;
    pcl::PointCloud<pcl::PointXYZRGB> recent_cloud;
    pcl::PointCloud<pcl::PointXYZRGB> new_cloud; };

struct CloudProperties {
    pcl::PointCloud<pcl::PointXYZRGB> cloud;
    pcl::KdTreeFLANN<pcl::PointXYZRGB> kdtree;

    void initialize() {
          pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloudPtr(new pcl::PointCloud<pcl::PointXYZRGB>);
          *cloudPtr = cloud;
          kdtree.setInputCloud(cloudPtr);
    }};


class ClusterAssociation {
public:
    ClusterAssociation(ros::NodeHandle& private_nh_) : max_freq_(1000), static_association_switch(false), radius(0.2),
tfListener_(tfBuffer_)
    {
          clusters_sub_.subscribe(nh_, "cloud_clusters", 10);
          clusters_sub_.registerCallback(boost::bind(&ClusterAssociation::saveCluster, this, _1));

          seeds_sub_.subscribe(nh_, "seed_transforms", 10);
          seeds_sub_.registerCallback(boost::bind(&ClusterAssociation::saveSeeds, this, _1));

          switch_sub_.subscribe(nh_, "static_association_switch" ,10);
          switch_sub_.registerCallback(boost::bind(&ClusterAssociation::switchAssoc, this, _1));

          private_nh_.getParam("grid_cells_x", gx);
          private_nh_.getParam("grid_cells_y", gy);
          private_nh_.getParam("grid_cells_z", gz);
          private_nh_.getParam("xmax", xmax);
          private_nh_.getParam("xmin", xmin);
          private_nh_.getParam("ymax", ymax);
          private_nh_.getParam("ymin", ymin);
          private_nh_.getParam("zmax", zmax);
          private_nh_.getParam("zmin", zmin);


/*        gx = 50;
          gy = 50;
          gz = 20;
          xmax = 2.5;
```

```
        ymax = 2.5;
        zmax = 1.3;
        xmin = -2.5;
        ymin = -2.5;
        zmin = 0.01;
*/

        c = s = false;
        timer_ = nh_.createTimer(ros::Duration(1.0/max_freq_), boost::bind(&ClusterAssociation::onTimer, this, _1));

        associated_clusters_pub_ = nh_.advertise<ACVector>("associated_clusters", 5);}
//      associated_clusters_pub_ = nh_.advertise<sensor_msgs::PointCloud2>("associated_clusters", 5);}


    void onTimer(const ros::TimerEvent& e) {
        if (c && s) {
                AssociateSeedsWithClusters();
                publishSeeds();
                c = false;
                s = false;
        }
    }

    void publishSeeds() {
        ACVector clusterObject;
        for(auto& x : temporary_seeds_) {
                for(auto& y : x.second) {
               if(y.second.frame == "")
                        continue;
                        ACluster object;
                        make_cloud_points(y.second.new_cloud, object.cloudpoints);
               object.transform = y.second.new_transform;
               object.label = y.second.label;
               object.frame = y.second.frame;
               object.object = y.second.object;
               clusterObject.clusters.push_back(object);
                }
        }
       clusterObject.gx = gx;
       clusterObject.gy = gy;
       clusterObject.gz = gz;
       clusterObject.xmax = xmax;
       clusterObject.ymax = ymax;
       clusterObject.zmax = zmax;
       clusterObject.xmin = xmin;
       clusterObject.ymin = ymin;
       clusterObject.zmin = zmin;
       associated_clusters_pub_.publish(clusterObject);
    }


    void make_cloud_points(pcl::PointCloud<pcl::PointXYZRGB>& cloud, cpoints& points) {
        geometry_msgs::Point point;
        for(int i = 0; i < cloud.points.size(); i++) {
                point.x = cloud.points[i].x;
                point.y = cloud.points[i].y;
                point.z = cloud.points[i].z;
                points.cloudpoints.push_back(point);
        }
    }

    void switchAssoc(const std_msgs::Int8::ConstPtr msg) {
        if(msg->data == 0) static_association_switch = false;
        if(msg->data == 1) static_association_switch = true;}

    void saveCluster(const Clouds::ConstPtr& cloudvec) {
        clusters_lock_.lock();
        clouds_.clear();                                //Clear old clouds whenever new ones arrive
        pcl::PointCloud<pcl::PointXYZRGB> temp;
        CloudProperties cp;
        for(auto x : cloudvec->clouds) {
                pcl::fromROSMsg(x, temp);
                cp.cloud = temp;
                cp.initialize();
                clouds_.push_back(cp);      //Vector updated with new clouds
        }
        c = true;
```

```
        clusters_lock_.unlock();}

    void saveSeeds(const Seeds::ConstPtr& seedvec) {
        seeds_lock_.lock();
        Seed temp;
        std::vector<std::string> frames;

        for(int i{0}; i < seedvec->transforms.size(); i++) {
                temp.label = seedvec->type[i];
                temp.frame = seedvec->frame[i];
                temp.object = seedvec->object[i];
                frames.push_back(seedvec->frame[i]);

                if(static_association_switch == false && temp.label == "static") // Update the seeds
                {
                        temp.new_transform = seedvec->transforms[i];
                        temp.initial_transform = temp.new_transform;
                        temp.recent_transform = temp.new_transform;
                        temporary_seeds_[temp.label][temp.frame] = temp;
                }
                if(static_association_switch == true && temp.label == "static") {} // Don't updat the seed transforsm
                if(static_association_switch == false && temp.label == "dynamic")
                {
                        temp.new_transform = seedvec->transforms[i];
                        temp.initial_transform = temp.new_transform;
                        temp.recent_transform = temp.new_transform;
                        temporary_seeds_[temp.label][temp.frame] = temp;                // Update the seeds
                }
                if(static_association_switch == true && temp.label == "dynamic")
                {
                        std::unordered_map<std::string,Seed>::const_iterator got =
temporary_seeds_["dynamic"].find(seedvec->frame[i]);
                        if(got == temporary_seeds_["dynamic"].end()) {

                                // If dynamic seed appeared only after static association
                                temp.new_transform = seedvec->transforms[i];
                                temp.initial_transform = temp.new_transform;
                                temp.recent_transform = temp.new_transform;
                                temporary_seeds_[temp.label][temp.frame] = temp;    // Map the seed and store its
current transform once
                        }
                        else {
                                temp.new_transform = seedvec->transforms[i];            // Otherwise dynamic seed
already existed before static association
                                temporary_seeds_[temp.label][temp.frame].new_transform = temp.new_transform; //
Update only the transforms in the seeds
                        }
                }
        }

        // Also check that if there are any Frames in the saved dictionary that are no longer in the screen, then
delete those entries.

        std::vector<std::string> tempstring;
        for(auto x : temporary_seeds_["dynamic"]) {
                if (std::find(frames.begin(),frames.end(), x.first) == frames.end()) {
                        std::cout<<"\nFrame "<<x.first<<" not found in list of seeds.";
                        tempstring.push_back(x.first);
                }
        }

        for(auto x : tempstring)
                temporary_seeds_["dynamic"].erase(x);

        s = true;
        seeds_lock_.unlock();
    }

    int rSearch(geometry_msgs::TransformStamped& x, CloudProperties& y) {
        pcl::PointXYZRGB searchPoint;
        searchPoint.x = x.transform.translation.x;
        searchPoint.y = x.transform.translation.y;
        searchPoint.z = x.transform.translation.z;
        std::vector<int> pointIdxRadiusSearch;
        std::vector<float> pointRadiusSquaredDistance;
        int returnvalue = y.kdtree.radiusSearch(searchPoint, radius, pointIdxRadiusSearch, pointRadiusSquaredDistance);
        return returnvalue;
```

```
    }


    int rSearch(geometry_msgs::TransformStamped& x, pcl::PointCloud<pcl::PointXYZRGB>& y) {
          // Overloaded Function
          pcl::PointXYZRGB searchPoint;
          CloudProperties temp;
          if(y.points.size() == 0)
                    return 0;
          temp.cloud = y;
          temp.initialize();
          searchPoint.x = x.transform.translation.x;
          searchPoint.y = x.transform.translation.y;
          searchPoint.z = x.transform.translation.z;
          std::vector<int> pointIdxRadiusSearch;
          std::vector<float> pointRadiusSquaredDistance;
          int returnvalue = temp.kdtree.radiusSearch(searchPoint, radius, pointIdxRadiusSearch,
pointRadiusSquaredDistance);
          return returnvalue;
    }

    void rSearch(pcl::PointXYZRGB& search_point, pcl::PointCloud<pcl::PointXYZRGB>& input_cloud,
pcl::PointCloud<pcl::PointXYZRGB>& output_cloud) {
          // Overloaded Function
          CloudProperties temp;
          if(input_cloud.points.size() == 0)
                    return;
          temp.cloud = input_cloud;
          temp.initialize();
          std::vector<int> pointIdxRadiusSearch;
          std::vector<float> pointRadiusSquaredDistance;
          int returnvalue = temp.kdtree.radiusSearch(search_point, radius, pointIdxRadiusSearch,
pointRadiusSquaredDistance);
          if (returnvalue < 1) // If the point has no other point in the input point cloud in its vicinity, add it to the
output cloud
                    output_cloud.points.push_back(search_point);
    }



    void AssociateSeedsWithClusters() {
          if(static_association_switch == false) {
                    for(auto& x : temporary_seeds_["static"]) {
                              if(x.second.object.compare("Target") != 0) {
                                        for(auto& y : clouds_) {
                                                  if(rSearch(x.second.initial_transform, y) > 0) {
                                                            x.second.initial_cloud = y.cloud;
                                                            x.second.recent_cloud = y.cloud;
                                                            x.second.new_cloud = y.cloud;
                                                  }
                                        }
                                        //std::cout<<"Currently Seed : "<<x.second.frame<<" and Cloudpoints :
"<<x.second.new_cloud.points.size()<<std::endl;

                              }
                    }

                    for(auto& x : temporary_seeds_["dynamic"]) {
                              if(x.second.object.compare("Target") != 0) {
                                        for(auto& y : clouds_) {
                                                  if(rSearch(x.second.initial_transform, y) > 0) {
                                                            x.second.initial_cloud = y.cloud;
                                                            x.second.recent_cloud = y.cloud;
                                                            x.second.new_cloud = y.cloud;

                                                  }
                                        }
                                        std::cout<<"Currently static association is False, Dynamic Seed :
"<<x.second.frame<<" and Cloudpoints : "<<x.second.new_cloud.points.size()<<std::endl;

                              }
                              else {}
                    }
          }

          else {
```

```
std::cout<<std::endl<<std::endl<<"========================================================"<<std::endl;
                    // Static Association switch is now True, so static seeds are saved and dynamic seeds have initial
clouds wherever possible.
                    for(auto& x : temporary_seeds_["static"]) {
                              if(x.second.object.compare("Target") != 0) {
                                        //std::cout<<"\nCurrently Static Seed : "<<x.second.frame;


                                        if(x.second.initial_cloud.width*x.second.initial_cloud.height!=0) {
                                                  transforms::doTransform(x.second.initial_cloud, x.second.new_cloud,
x.second.new_transform, x.second.initial_transform);
                                                  //std::cout<<" Initial cloud present, so transforming. Cloudpoints :
"<<x.second.new_cloud.points.size()<<std::endl;
                                        }
                                        else{
                                                  for(auto& y : clouds_) {
                                                            if(rSearch(x.second.initial_transform, y) > 0) {
                                                                      x.second.initial_cloud = y.cloud;
                                                                      x.second.recent_cloud = y.cloud;
                                                                      x.second.new_cloud = y.cloud;
                                                            }
                                                  }
                                                  //std::cout<<"Initial cloud not present, so finding association.
Cloudpoints : "<<x.second.new_cloud.points.size()<<std::endl;
                                        }
                              }
                              else {}
                    }


                    for(auto& x : temporary_seeds_["dynamic"]) { // For each seed
                              if(x.second.object.compare("Target") != 0) {
                                        std::cout<<"\nCurrently Dynamic Seed : "<<x.second.frame;

                                        std::cout<<"\nCurrently object type is : "<<x.second.object;

                                        for(auto& y : clouds_) {                        // Go through each cluster
                                                  if(rSearch(x.second.new_transform, y) > 0) { // If the new seed has a new
cluster associated with it
                                                            x.second.new_cloud = y.cloud;                    // save as the
new cloud
                                                            std::cout<<" found an association :
"<<x.second.new_cloud.points.size()<<std::endl;
                                                  }
                                        }
                              }
                              else {}
                    }

                    // Now checking if there are multiple associations for dynamic seeds only
                    for(auto& x : temporary_seeds_["dynamic"]) {
                              if(x.second.object.compare("Target") != 0) {
                                        x.second.flag = false;
                                        std::cout<<"\nCurrently Dynamic Seed : "<<x.second.frame;


                                        for(auto& y : temporary_seeds_) {
                                                  for(auto& z : y.second) {
                                                            if(z.second.frame != x.second.frame &&
rSearch(z.second.new_transform, x.second.new_cloud) > 0) {
                                                                      x.second.flag = true; // If any other new seed is
contained in the new cluster, set the flag to true.
                                                                      std::cout<<" has flag set to :
"<<x.second.flag<<std::endl;
                                                            }
                                                  }
                                        }
                              }
                    }


                    for(auto& x : temporary_seeds_["dynamic"]) {
                              if(x.second.object.compare("Target") != 0) {
                                        std::cout<<"\nCurrently Dynamic Seed : "<<x.second.frame;
```

```
                                    if(x.second.new_cloud.width*x.second.new_cloud.height != 0 && x.second.flag ==
false) {    // If no other new seeds associate with this new cluster, save as recent cloud and recent transform
                                        std::cout<<" and there is no other association"<<std::endl;
                                        x.second.recent_cloud = x.second.new_cloud; // Recent cloud is stored
only when there is no other association
                                        x.second.recent_transform = x.second.new_transform;
                                        std::cout<<"Setting recent cloud :
"<<x.second.recent_cloud.points.size();
                                        if(x.second.initial_cloud.width*x.second.initial_cloud.height == 0)    {
// if there exists no initial cloud, save as initial cloud
                                            x.second.initial_cloud = x.second.new_cloud;
                                            x.second.initial_transform = x.second.new_transform;
                                            std::cout<<", setting initial cloud :
"<<x.second.initial_cloud.points.size()<<std::endl;
                                        }
                                        else {}
                                    }
                                    else if(x.second.new_cloud.width*x.second.new_cloud.height != 0 && x.second.flag ==
true) { // There are other associated seeds.
                                        std::cout<<" but there is another association."<<std::endl;
                                        if(x.second.recent_cloud.width*x.second.recent_cloud.height != 0) { //
There exists a recent cloud, which means earlier it was unique associated
                                            transforms::doTransform(x.second.recent_cloud,
x.second.new_cloud, x.second.new_transform, x.second.recent_transform);
                                            std::cout<<" Using a recent cloud if it exists :
"<<x.second.new_cloud.points.size();
                                        }
                                        else { // There exists no recent cloud
                                            if(x.second.initial_cloud.width*x.second.initial_cloud.height !=
0) { // Is there an initial cloud that was unique associated
                                                transforms::doTransform(x.second.initial_cloud,
x.second.new_cloud, x.second.new_transform, x.second.initial_transform);
                                                 // transform the initial cloud to the new cloud
                                                std::cout<<", Using an initial cloud if it exists :
"<<x.second.new_cloud.points.size()<<std::endl;
                                            }
                                            else {
                                                std::cout<<"No initial or recent clouds, so gonna do
some subtractions. "<<std::endl;
                                                // But we have the new cloud for this particular seed.
We save this as a subtraction cloud.
                                                x.second.subtraction_cloud = x.second.new_cloud;
                                            }
                                        }
                                    }

                                    else if(x.second.new_cloud.width*x.second.new_cloud.height == 0) {    // If the
seed has no associated cluster as per newly acquired cloud
                                        std::cout<<" There was no associated cluster as per newly acquired cloud.
";
                                        if(x.second.recent_cloud.width*x.second.recent_cloud.height != 0) { //
There exists a recent cloud that was uniquely associated
                                            transforms::doTransform(x.second.recent_cloud,
x.second.new_cloud, x.second.new_transform, x.second.recent_transform);
                                            x.second.flag = false;
                                            std::cout<<"Found a recent cloud to use :
"<<x.second.new_cloud.points.size()<<std::endl;
                                        }
                                        else { // There exists no recent cloud
                                            if(x.second.initial_cloud.width*x.second.initial_cloud.height !=
0) { // Is there an initial cloud that was uniquely associated
                                                transforms::doTransform(x.second.initial_cloud,
x.second.new_cloud, x.second.new_transform, x.second.initial_transform);
                                                x.second.flag = false; // transform the initial cloud
to the new cloud
                                                std::cout<<"Found no recent cloud but found an initial
cloud to use : "<<x.second.new_cloud.points.size()<<std::endl;
                                            }
                                            else {
                                                std::cout<<"Found no recent nor initial cloud, and no
association with any new cloud. So setting nothing"<<std::endl;
                                                x.second.flag = false;
                                            }
                                        }
                                    }
                                }
                                else {}
```

```
                }


                // Now all seeds have associated new_clouds, and some seeds have subtraction clouds. This means that
for all new_clouds associated with
                // seeds in the multiple_associations set, we do an rSearch on the points in subtraction_cloud and
delete that point from the subtractioncloud.

                for(auto& x : temporary_seeds_["dynamic"]) {
                        if(x.second.object.compare("Target") != 0) {
                                // This search will only apply to dynamic seeds.
                                if(x.second.subtraction_cloud.points.size() != 0) {
                                        // Only if there exists a subtraction cloud
                                        // Make a new point cloud that is a copy of the subtraction cloud. Then
do copied_cloud.clear().
                                        // Then take each point of the subtraction cloud, do an rSearch on each
saved cloud. If rSearch returns none, save
                                        // that point by inserting into copied_cloud.
                                        // Because two objects cannot overlap in reality, the copied_cloud will
contain only those points that belong exclusively to
                                        // that object. So we need to write an rSearch that takes in a
PointXYZRGB and a PointCloud<T> as input, and also
                                        // returns PointCloud<T> as output.
                                        // It is possible that an object enters the scene with multiple
associations and that none of them have an initial or recent transform
                                        // Only in that case, one of the object's copy_cloud will be empty after
subtraction. So when we receive seeds in database.py,
                                        // If the seed has no clouds associated with it, it can mean one of two
things - that there are multiple associations, or that
                                        // the object is no longer in the scene. We can do a simple if-else check
to determine which situation it is.
                                        pcl::PointCloud<pcl::PointXYZRGB> copy_cloud;
                                        for(auto& y : temporary_seeds_) {
                                                for(auto& z : y.second) {
                                                        for(auto& point : x.second.subtraction_cloud.points) {
                                                                rSearch(point, z.second.new_cloud,
copy_cloud);
                                                        }
                                                }
                                        }
                                        x.second.new_cloud = copy_cloud;
                                        x.second.recent_cloud = copy_cloud;
                                        x.second.recent_transform = x.second.new_transform;
                                        x.second.initial_cloud = copy_cloud; // Because there was a subtraction
cloud means there was no initial nor recent cloud
                                        x.second.initial_transform =x.second.new_transform;
                                }
                        }
                        else {}
                }
        }
    }




private:
    ros::NodeHandle nh_;
    tf2_ros::Buffer tfBuffer_;
    tf2_ros::TransformListener tfListener_;
    ros::Timer timer_;

    double max_freq_, radius;
    bool c, s, static_association_switch;

    boost::mutex clusters_lock_, seeds_lock_;

    message_filters::Subscriber<Clouds> clusters_sub_;
    message_filters::Subscriber<Seeds> seeds_sub_;
    message_filters::Subscriber<std_msgs::Int8> switch_sub_;

    std::vector<CloudProperties> clouds_;

    std::unordered_map<std::string, std::unordered_map<std::string, Seed>> temporary_seeds_;
    // unordered_map<Key, T>::iterator it, (*it).first = key, second = value.
    // insert and access values using [] -> use for insertion only
```

```
    // umap.insert(make_pair(key, value))
    // umap.at(key) gives value - use for access and modification only

    double xmax, xmin, ymax, ymin, zmax, zmin;
    int gx, gy, gz;


    ros::Publisher associated_clusters_pub_;
};

// Main Function
int main(int argc, char** argv)
{
    ros::init(argc, argv, "cluster_assoc_node");

    ros::NodeHandle private_nh_("~");

    ClusterAssociation ca(private_nh_);
    ros::spin();
    return 0;
}

/*
Situations:
1) Initially no cloud associated with seed
2) Initially unique cloud associated with seed but only partially
3) Initially unique cloud associated with seed but fully
4) Initially non-unique clouds associated with seed
5) Newly acquired cloud has no association for the seed
6) Newly acquired cloud has unique association for the seed but only partially
7) Newly acquired cloud has unique association for the seed but fully
8) Newly acquired seed has non-unique association for the seed
9) Recently no cloud associated with seed
10) Recently unique cloud associated with seed but only partially
11) Recently unique cloud associated with seed but fully
12) Recently non-unique cloud associated with seed.

There should be 64 combinations but we can simplify this

Recent and New transforms always get updated in dynamic seeds. Init transforms are stationary.
Clouds get updated only after association.

If newly acquired cloud clusters around seed, check if it clusters around other seeds also.
    If not, set to new_cloud and recent_cloud, and recent transform
           If initial cloud exists, skip. Else set this as initial_cloud as well, and initial_transform as well.
           Set flag to "unique_cloud"
    If yes, check if recent cloud exists that does not cluster around other seeds
           If yes, transform the recent cloud and save it as new cloud
                    Set flag to "unique"
           If no, check if initial cloud exists that does not cluster around other seeds
                    If yes, transform initial cluod to new_cloud
                              Set flag to "unique"
                    If no, store the new_cloud as new_cloud and set the flag to "subtract".
If no newly acquired cloud clusters around seed, check if there is a recent cloud that does not cluster around other
seeds
    If yes, transform recent cloud to new cloud and set flag to "unique"
    If no, check if there is an initial cloud that does not cluster around other seeds
           If yes, transform initial cloud to new cloud and set flag to "unique"
           If no, do nothing

*/
```

C++ Code for Database : Occupancy Grid Map generation

Path: ankurjay/ankurbot/ankurbot_description/src/database.cpp

```
// Headers
#include <ros/ros.h>

#include <algorithm>
#include <vector>
#include <string>
#include <tuple>
#include <unordered_map>

// ROS Messages
#include <geometry_msgs/TransformStamped.h>
#include <std_msgs/Int8.h>
#include <visualization_msgs/Marker.h>
#include <geometry_msgs/Point.h>

// Custom Message
#include "ankurbot_multisim/AssociatedCluster.h"
#include "ankurbot_multisim/AssociatedClusterVector.h"
#include "ankurbot_multisim/CloudPoints.h"
#include "ankurbot_multisim/MapMetaData.h"
#include "ankurbot_multisim/GridIndex.h"
#include "ankurbot_multisim/GridIndexVector.h"

template <class T>
class Grid3D {
public:

    Grid3D(int rowwid = 0, int colwid = 0, int stackwid = 0) {
          rows = rowwid;
          cols = colwid;
          stacks = stackwid;
          long_form.reserve(rows*cols*stacks);
    }

    T& operator() (int a, int b, int c) {
          // X,Y,Z = Row, Column, Stack
          // In X direction (+ve right) there are columns of stacks. The columns are in Y direction (+ve north) and
stacks are in Z direction (+ve up)
          // In Y direction (+ve north) there are stacks of cells, each stack has a height along Z direction (+ve up)
          // In Z direction (+ve up) there are cells
          return long_form[cols*stacks*a + stacks*b + c];
    }

    void default_value(T a) {
          long_form = std::vector<T>(rows*cols*stacks, a);
    }

    std::vector<T> get_vector() {
          return long_form;
    }

private:
    std::vector<T> long_form;
    int rows;
    int cols;
    int stacks;
};


class Database {
public:
    Database()
    {
          cluster_sub_ = nh_.subscribe("associated_clusters", 5, &Database::callback, this);
          metadata_pub_ = nh_.advertise<ankurbot_multisim::MapMetaData>("planning_metadata", 5);
          collisions_pub_ = nh_.advertise<visualization_msgs::Marker>("collisions", 5);

    }

    void callback(const ankurbot_multisim::AssociatedClusterVector::ConstPtr& msg) {
          init_params(*msg);
```

```
            store_clouds(*msg);
            put_clouds_in_bins();
            publish_current_state();
            publish_collisions();
            //print_database();
    }

    void init_params(const ankurbot_multisim::AssociatedClusterVector& data) {
            x_cells = data.gx;
            y_cells = data.gy;
            z_cells = data.gz;
            xmax = data.xmax;
            xmin = data.xmin;
            ymax = data.ymax;
            ymin = data.ymin;
            zmax = data.zmax;
            zmin = data.zmin;
            resolution_x = (xmax - xmin)/static_cast<double>(x_cells);
            resolution_y = (ymax - ymin)/static_cast<double>(y_cells);
            resolution_z = (zmax - zmin)/static_cast<double>(z_cells);
            grid = Grid3D<unsigned int>(x_cells, y_cells, z_cells);
            grid.default_value(0);

            bins = Grid3D<std::unordered_map<std::string, std::vector<std::tuple<double, double, double>>>>(x_cells,
y_cells, z_cells);
            std::unordered_map<std::string, std::vector<std::tuple<double, double, double>>> zeros;
            bins.default_value(zeros);

            colliding_check_cells.clear();
    }

    void store_clouds(const ankurbot_multisim::AssociatedClusterVector& data) {
            // Store the cloudpoints for the object
            database_cloudpoints.clear();
            database_object_type.clear();
            database_seed_transform.clear();
            for(auto& x : data.clusters) {
                    if(database_UID.find(x.frame) == database_UID.end()) {                                                  // If
the UID of the Debris, Robot, obstacle or target is not available, create a UID
                            int UID = (rand()%256)*1000000 + (rand()%256)*1000 + rand()%256;
                            database_UID[x.frame] = UID;
                    }
                    else { }
                    database_cloudpoints[x.frame] = point_collection(x.cloudpoints.cloudpoints);
                    database_object_type[x.frame] = x.object;
                    database_seed_transform[x.frame] = x.transform;



            }
    }

    std::vector<std::tuple<double, double, double>> point_collection(const std::vector<geometry_msgs::Point>&
cloudpoints) {
            std::vector<std::tuple<double, double, double>> collection;
            for(auto& x : cloudpoints)
                    collection.push_back(std::tuple<double, double, double>(x.x, x.y, x.z));
            return collection;
    }

    void put_clouds_in_bins() {
            database_grid_idxs.clear();
            for(auto& x : database_cloudpoints) {
                    // x.first is the frame name and x.second is the vector of tuples of points

                    std::set<std::tuple<int, int, int>> set_of_indices;
                    for(auto& point : x.second){

                            // Get the Grid Indices for each point
                            if (std::get<0>(point) >= xmin && std::get<0>(point) < xmax && std::get<1>(point) >= ymin &&
std::get<1>(point) < ymax && std::get<2>(point) >= zmin && std::get<2>(point) < zmax) {
                                    int ix = static_cast<int>((std::get<0>(point) - xmin)/resolution_x);
                                    int iy = static_cast<int>((std::get<1>(point) - ymin)/resolution_y);
                                    int iz = static_cast<int>((std::get<2>(point) - zmin)/resolution_z);
                                    set_of_indices.insert(std::make_tuple(ix, iy, iz));

                                    // Fill up the bins with the cloud points
                                    if (bins(ix, iy, iz).empty()) {
```

```
                                        std::unordered_map<std::string, std::vector<std::tuple<double, double,
double>>> dictionary;
                                        dictionary[x.first].push_back(point);
                                        bins(ix, iy, iz) = dictionary;
                                }
                                else
                                        bins(ix, iy, iz)[x.first].push_back(point);

                        }
                        else {}
                }

                // Save the tuple of grid indices for that object
                database_grid_idxs[x.first] = set_of_indices;

                // Now fill up the occupancy grid
                for(auto& element : set_of_indices) {
                        if(grid(std::get<0>(element), std::get<1>(element), std::get<2>(element)) == 0)
                                grid(std::get<0>(element), std::get<1>(element), std::get<2>(element)) = 1;
                        else if(grid(std::get<0>(element), std::get<1>(element), std::get<2>(element)) == 1) {
                                grid(std::get<0>(element), std::get<1>(element), std::get<2>(element)) = 2;
                                colliding_check_cells.insert(element);
                        }
                }
        }

        // Finally, check collisions
        check_collisions();
    }

    void check_collisions() {
        if(colliding_check_cells.size()!=0)
                std::cout<<"\nProximity for Collision, Beware!";
        for(auto& item : colliding_check_cells) {
                //std::cout<<"("<<std::get<0>(item)<<","<<std::get<1>(item)<<","<<std::get<2>(item)<<")"<<std::endl;
        }
        // GJK algorithm implementation
        // Choose a direction D
        // Compute the support. The support computes the dot product D.B for a point in B.
        // Find the max((D.B)) i.e. support(D,B) and put that B into empty list
        // Now D = -D
        // Then we find max(-D.A) i.e support (D,A) and we put that A into the empty list
        // If -D.A < 0 it means that the projection of A on D is not towards the origin but away from the origin. So A
will never reach origin.
        // If not, then we add A to the list with B and we do a simplex routine Simplex([list], -D)
        // The Simplex routine will either detect an intersection and quit the loop, or will continue the loop of
adding points to the [list] and doing stuff
        // The [list] will have max of 4 points
        // In the Simplex, go through each point in [list] and find which one is the closest to the origin.
        // The smallest shape that we have to deal with in a Simplex, is a line
        // Now we know B was already there in [list] and A was what was added
        // There are 3 possible results for a line : Either B is closest to origin, or A is closest to origin, or
origin lies somewhere in between
        // The origin does not lie near B because the very reason 'A' was selected was because it was the farthest
point in the direction of the origin
        // So either A is closer to the origin, or the origin lies between A and B.
        // If you do B-A and O-A we get AB (pointing towards B) and AO (pointing towards O) vectors. If AB(dot)AO > 0
then origin is in between A and B becase
        // that means the component of AO lies along AB. If so, the Simplex([list], -D) will return
AB(cross)AO(cross)AB as the new search direction vector
        // Otherwise return AO as the new search direction vector
        //


    }

    void publish_current_state() {
        ankurbot_multisim::MapMetaData metadata;
        metadata.gx = x_cells;
        metadata.gy = y_cells;
        metadata.gz = z_cells;
        metadata.xmax = xmax;
        metadata.xmin = xmin;
        metadata.ymax = ymax;
        metadata.ymin = ymin;
        metadata.zmax = zmax;
        metadata.zmin = zmin;
```

```
        for(auto& x : database_UID) {
                metadata.frames.push_back(x.first);
                metadata.UIDs.push_back(database_UID[x.first]);
                metadata.objects.push_back(database_object_type[x.first]);
                metadata.seed_poses.push_back(database_seed_transform[x.first]);
                ankurbot_multisim::GridIndexVector idxvec;
                for(auto& tuple : database_grid_idxs[x.first]) {
                        ankurbot_multisim::GridIndex idx;
                        idx.x = std::get<0>(tuple);
                        idx.y = std::get<1>(tuple);
                        idx.z = std::get<2>(tuple);
                        idxvec.indices.push_back(idx);
                }
                metadata.grid_indices.push_back(idxvec);
        }
        metadata.grid = grid.get_vector();
        std::vector<ankurbot_multisim::GridIndex> colls(colliding_check_cells.size());
        for(auto& x : colliding_check_cells) {
                ankurbot_multisim::GridIndex gi;
                gi.x = std::get<0>(x);
                gi.y = std::get<1>(x);
                gi.z = std::get<2>(x);
                colls.push_back(gi);
        }
        metadata.collisions = colls;
        metadata_pub_.publish(metadata);
    }

    void predict_new_state() { // Most likely this will be a service callback
        // Take the names of the frames and their new positions
        // transform each stored cloud into new positions and bin them again as predicted_bins
        // Check for collisions by only looking at those bins where there are clouds from more than one entity
        // If no collisions, send a no-collision flag, but the grid cells are marked red
        // Otherwise, send a collsion flag and mark the cells red. The planner will take care of the collision flag. If
collision is imminent with a target,
        // the planner will allow that motion.
    }

    void print_database() {
        for(auto& x : database_UID) {
                std::cout<<"\nFrame : "<<x.first;

                std::cout<<"\nGrid Indexes : [";
                for(auto& x : database_grid_idxs[x.first]) {
                        std::cout<<"("<<std::get<0>(x)<<","<<std::get<1>(x)<<","<<std::get<2>(x)<<")";
                }
                std::cout<<"]"<<std::endl;

                std::cout<<"\nObject Type : "<<database_object_type[x.first];

                std::cout<<"\nUID : "<<database_UID[x.first];
                std::cout<<"\n====================";

        }}

    void publish_collisions() {
        visualization_msgs::Marker marker;
        marker.header.frame_id = "world";
        marker.ns = "collisions";
        marker.id = 999999999;
        marker.type = visualization_msgs::Marker::CUBE_LIST;
        marker.action = visualization_msgs::Marker::ADD;
        marker.color.r = 1.0;
        marker.color.g = 0.0;
        marker.color.b = 0.0;
        marker.color.a = 1.0;
        marker.pose.position.x = 0;
        marker.pose.position.y = 0;
        marker.pose.position.z = 0;
        marker.pose.orientation.x = 0;
        marker.pose.orientation.y = 0;
        marker.pose.orientation.z = 0;
        marker.pose.orientation.w = 0;

        marker.scale.x = resolution_x;
        marker.scale.y = resolution_y;
        marker.scale.z = resolution_z;
```

```
            marker.points = get_centroids(colliding_check_cells);
            collisions_pub_.publish(marker);}

    std::vector<geometry_msgs::Point> get_centroids(std::set<std::tuple<int, int, int>>& input) {
            std::vector<geometry_msgs::Point> centroids;
            for(auto& x : input) {
                    geometry_msgs::Point point;
                    point.x = xmin + std::get<0>(x)*resolution_x + 0.5*resolution_x;
                    point.y = ymin + std::get<1>(x)*resolution_y + 0.5*resolution_y;
                    point.z = zmin + std::get<2>(x)*resolution_z + 0.5*resolution_z;
                    centroids.push_back(point);
            }
            return centroids;
    }

private:
    ros::NodeHandle nh_;

    int x_cells;
    int y_cells;
    int z_cells;
    double xmax;
    double xmin;
    double ymax;
    double ymin;
    double zmax;
    double zmin;
    double resolution_x;
    double resolution_y;
    double resolution_z;

    Grid3D<unsigned int> grid;
    Grid3D<std::unordered_map<std::string, std::vector<std::tuple<double, double, double>>>> bins;

    std::unordered_map<std::string, std::vector<std::tuple<double, double, double>>> database_cloudpoints;
    std::unordered_map<std::string, std::string> database_object_type;
    std::unordered_map<std::string, int> database_UID;
    std::unordered_map<std::string, geometry_msgs::TransformStamped> database_seed_transform;
    std::unordered_map<std::string, std::set<std::tuple<int, int, int>>> database_grid_idxs;

    std::set<std::tuple<int, int, int>> colliding_check_cells;

    ros::Subscriber cluster_sub_;
    ros::Publisher metadata_pub_;
    ros::Publisher collisions_pub_;
};

// Main Function
int main(int argc, char** argv)
{
    ros::init(argc, argv, "database_node");
    Database db;
    ros::spin();
    return 0;
}

// Use std::get<idx>(tuple_name) to get value of item in tuple
// std::make_tuple(items) returns the tuple
// std::tuple<type1, type2, type3...>(items) also returns a tuple
// std::set<type>
// set.clear(), set.insert()
// if(set.find(element)!=set.end()) cout<<*(set.find(element))
// for(auto it = set.begin(), it!= set.end(); ) if *it == element, it = set.erase(it) else ++it; ----- erase from set
```

APPENDIX C

Note: The code provided here is part of a larger repository, and only reflects the task of immediate interest. For reference, the repository can be accessed at: https://github.com/ankurjay/ankurbot/tree/multi_processing_trials_skidsteer_astar

Relative paths are provided per this repository, for reference.

Python Code for Multi Robot Controls Server

Path: ankurjay/ankurbot/ankurbot_control/scripts/multi_robot_controls_server.py

```
import roslib; roslib.load_manifest('ankurbot_control') #Load the package.xml of the relevant package
import rospy, math
import sys # Needed for main function argument sys.argv
import numpy as np
from ankurbot_multisim.msg import TransformStampedVector, MoveRobotAction, MoveRobotFeedback, MoveRobotResult
from geometry_msgs.msg import Twist, Pose, Transform, TransformStamped
from control_msgs.msg import JointControllerState
from std_msgs.msg import Int8, Float64
import actionlib
from tf.transformations import euler_from_quaternion, quaternion_from_euler
from sensor_msgs.msg import Joy

"""
This is a set of servers for each robot, it receives target poses from the client, and it
sends a feedback about current velocity and pose and completion
Receives message from seed_tf_publisher to keep track of frames. Don't want to waste time using
ActionServer instead of SimpleActionServer

The aim of having an action server is to receive the names of the frames of all robots in the scene and give them
cmd_vels or other commands.
So this is not only an ankurbot controller, this is for any robot.
Goal:
/ankurbot_1/rear_cmd_vel
/ankurbot_1/front_cmd_vel
/ankurbot_1/fl_position_controller/command
/ankurbot_1/fl_position_controller/state
"""

class AnkurbotController:
    def __init__(self, frame):
          self.server = actionlib.SimpleActionServer(frame + '/move_robot',MoveRobotAction, self.execute, False)
          self.feedback = MoveRobotFeedback()
          self.result = MoveRobotResult()
          self.flag = 'Not OK'
          self.frame = frame

          self.pub_drive = rospy.Publisher(frame + '/cmd_vel', Twist, queue_size=1)

          self.pub_fl = rospy.Publisher(frame + '/fl_position_controller/command', Float64, queue_size = 1)
          self.pub_fr = rospy.Publisher(frame + '/fr_position_controller/command', Float64, queue_size = 1)
          self.pub_rl = rospy.Publisher(frame + '/rl_position_controller/command', Float64, queue_size = 1)
          self.pub_rr = rospy.Publisher(frame + '/rr_position_controller/command', Float64, queue_size = 1)

          self.sub_fl = rospy.Subscriber(frame + '/fl_position_controller/state', JointControllerState,
self.check_wheels_turned)
          self.sub_fr = rospy.Subscriber(frame + '/fr_position_controller/state', JointControllerState,
self.check_wheels_turned)
          self.sub_rl = rospy.Subscriber(frame + '/rl_position_controller/state', JointControllerState,
self.check_wheels_turned)
          self.sub_rr = rospy.Subscriber(frame + '/rr_position_controller/state', JointControllerState,
self.check_wheels_turned)

          self.max_speed = 1 # I think this is m/s
          self.max_turn = 1 # This is rad/s

          # To rotate about body center
```

```
            self.ne1 = math.radians(45)
            self.nw1 = math.radians(-45)
            self.se1 = math.radians(45)
            self.sw1 = math.radians(-45)


            # To rotate about end effector
            self.ne2 = math.radians(-82.57)
            self.nw2 = math.radians(82.57)
            self.se2 = math.radians(84.09)
            self.sw2 = math.radians(-84.09)

            self.linvel = 0
            self.rotvel = 0



            self.current_pose = None
            self.last_pose = None
            self.transform_rate = 1

            self.server.start()

    def set_current_pose(self, pose):
            if self.current_pose is None:
                    self.current_pose = TransformStamped()
                    self.current_pose = self.ankurbot_pose_correction(pose)
            else:
                    if self.last_pose is None:
                            self.last_pose = TransformStamped()
                            self.last_pose = self.current_pose
                            self.current_pose = self.ankurbot_pose_correction(pose)
                    else:
                            if rospy.Time(self.last_pose.header.stamp.secs,
self.last_pose.header.stamp.nsecs)!=rospy.Time(self.current_pose.header.stamp.secs,
self.current_pose.header.stamp.nsecs):
                                    self.transform_rate = rospy.Time(self.current_pose.header.stamp.secs,
self.current_pose.header.stamp.nsecs).to_sec() - rospy.Time(self.last_pose.header.stamp.secs,
self.last_pose.header.stamp.nsecs).to_sec()
                                    #print("\n--- Transform receiving rate is now : ", self.transform_rate)
                            self.last_pose = self.current_pose
                            self.current_pose = self.ankurbot_pose_correction(pose)



            #print("\n---Current Pose saved is : ", math.degrees(self.get_z_angle(self.get_current_pose())))
            #print("\n---Current Pose saved is : ", (self.get_current_pose().transform.translation.x,
self.get_current_pose().transform.translation.y,self.get_current_pose().transform.translation.z))

    def ankurbot_pose_correction(self, pose):
            (r,p,y) = euler_from_quaternion((pose.transform.rotation.x, pose.transform.rotation.y,
pose.transform.rotation.z, pose.transform.rotation.w))
            (pose.transform.rotation.x, pose.transform.rotation.y, pose.transform.rotation.z, pose.transform.rotation.w) =
quaternion_from_euler(r, p, y+math.radians(90))
            return pose

    def get_current_pose(self):
            return self.current_pose

    def check_wheels_turned(self, data):
          epsilon = 0.01
          if data.process_value > data.set_point-epsilon and data.process_value < data.set_point+epsilon:
          self.flag = 'OK'
          else:
          self.flag = 'Not OK'

    def publish_twist(self, speed, turn, vel, rot):
            twist = Twist()
            twist.linear.x = vel*speed
            twist.angular.z = rot*turn
            self.pub_drive.publish(twist)

    def publish_angle_bindings(self, flag, angle=None):
            if angle is not None:
                    angleBindings = (angle, angle, angle, angle) # For translation or resetting wheels
            else:
                    if flag == False:
```

```
                        angleBindings = (self.ne2, self.nw2, self.sw2, self.se2) # ROtate about end effector
                elif flag == True:
                        angleBindings = (self.ne1, self.nw1, self.se1, self.sw1) # Rotate about body centroid


        self.pub_fl.publish(angleBindings[0])
        self.pub_fr.publish(angleBindings[1])
        self.pub_rl.publish(angleBindings[2])
        self.pub_rr.publish(angleBindings[3])

                while(self.flag == 'Not OK'): # Wait for flag to become OK
                        pass

        print("\n--- All wheels have turned to their positions")
        rospy.sleep(self.transform_rate)

    def angle_difference(self, angle1, angle2):
        angle = (angle1 - angle2)%(2*math.pi)
        if angle > math.pi:
                angle -= 2*math.pi
        return angle

    def get_z_angle(self, pose):
        if isinstance(pose, TransformStamped):
                return euler_from_quaternion((pose.transform.rotation.x, pose.transform.rotation.y,
pose.transform.rotation.z, pose.transform.rotation.w))[2]
        elif isinstance(pose, Pose):
                return euler_from_quaternion((pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w))[2]

    def get_y_angle(self, pose):
        if isinstance(pose, TransformStamped):
                return euler_from_quaternion((pose.transform.rotation.x, pose.transform.rotation.y,
pose.transform.rotation.z, pose.transform.rotation.w))[1]
        elif isinstance(pose, Pose):
                return euler_from_quaternion((pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w))[1]

    def get_x_angle(self, pose):
        if isinstance(pose, TransformStamped):
                return euler_from_quaternion((pose.transform.rotation.x, pose.transform.rotation.y,
pose.transform.rotation.z, pose.transform.rotation.w))[0]
        elif isinstance(pose, Pose):
                return euler_from_quaternion((pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w))[0]


    def position_difference_vector(self, targetpose, initpose):
        if isinstance(initpose, TransformStamped):
                (xi, yi, zi) = (initpose.transform.translation.x, initpose.transform.translation.y,
initpose.transform.translation.z)
        elif isinstance(initpose, Pose):
                (xi, yi, zi) = (initpose.position.x, initpose.position.y, initpose.position.z)
        if isinstance(targetpose, TransformStamped):
                (xt, yt, zt) = (targetpose.transform.translation.x, targetpose.transform.translation.y,
targetpose.transform.translation.z)
        elif isinstance(targetpose, Pose):
                (xt, yt, zt) = (targetpose.position.x, targetpose.position.y, targetpose.position.z)
        return (xi - xt, yi - yt, zi - zt) # REVIEW THIS

    def get_magnitude(self, distance_vector, bias = (1,1,0)):
        return math.sqrt(bias[0]*distance_vector[0]**2 + bias[1]*distance_vector[1]**2 + bias[2]*distance_vector[2]**2)

    def publish_feedback(self):
        (self.feedback.vx, self.feedback.vy, self.feedback.vz) = self.position_difference_vector(self.current_pose,
self.last_pose)
        self.feedback.ax = self.angle_difference(self.get_x_angle(self.last_pose), self.get_x_angle(self.current_pose))
        self.feedback.ay = self.angle_difference(self.get_y_angle(self.last_pose), self.get_y_angle(self.current_pose))
        self.feedback.az = self.angle_difference(self.get_z_angle(self.last_pose), self.get_z_angle(self.current_pose))
        self.feedback.time = self.transform_rate
        self.feedback.robotname = self.frame
        self.feedback.robottype = "Mobile"
        self.server.publish_feedback(self.feedback)

    def rotation_loop(self, angle, targetpose, rotationflag):

        self.publish_twist(0,0,0,0)
```

```
            rospy.sleep(5*self.transform_rate)
            self.publish_angle_bindings(False, 0)


            rotation = False

            if abs(math.degrees(angle)) > 1:
                     rotation = True
                     self.publish_angle_bindings(rotationflag)

            self.publish_feedback()

            while self.flag == 'Not OK': # Wait for wheels to turn
                     self.publish_feedback()
                     pass

            factor = 1

            while rotation is True:

                     if rotationflag == False:
                              turn = factor*self.max_turn
                     else:
                              turn = self.max_turn
                     speed = self.max_speed


                     if math.degrees(angle) > 0:
                              (vel, rot) = (0, 1)
                     elif math.degrees(angle) < 0:
                              (vel, rot) = (0, -1)

                     vel = vel*speed
                     rot = rot*turn

                     self.publish_twist(speed, turn, vel, rot)
                     rospy.sleep(self.transform_rate)

                     self.rotvel = self.angle_difference(self.get_z_angle(self.last_pose),
self.get_z_angle(self.current_pose))/self.transform_rate # This is in radians per second
                     #print("\n--- Rotational velocity is : ", math.degrees(self.rotvel), " deg/second")

                     z_angle = self.get_z_angle(self.current_pose)
                     #print("\n--- Current Pose's Z-angle is : ", z_angle)
                     angle = self.angle_difference(self.get_z_angle(targetpose), z_angle)

                     #print("\n--- Angle difference is now : ", math.degrees(angle))

                     #print("\n--- Product of rotvel and transform rate is : ", self.rotvel*self.transform_rate)
                     if abs(math.degrees(angle)) <= 2: # Apply Brakes
                              factor = 0.5*abs(math.degrees(angle))
                     elif abs(math.degrees(angle)) <= 30 and abs(math.degrees(angle)) > 2: # Apply Brakes
                              factor = 1
                     else:
                              factor = 5 # Go FULL SPEED!!!! WOO!!

                     self.publish_feedback()

                     if abs(math.degrees(angle)) < 1:
                              rotation = False

            self.publish_twist(0,0,0,0)
            rospy.sleep(5*self.transform_rate)
            self.publish_angle_bindings(False, 0)
            self.publish_feedback()

            print("\n--- Rotation complete") # THIS IS PERFECT, but maybe will develop a proportional controller later

        def translation_loop(self, targetpose, distance_offset_vector): # REVIEW THIS

            # The main issue here is that the wheel angles that I compute, are with respect to the vector between world and
ankurbot/ball's transform.
            # But the wheel controller works as per angles dictated w.r.t body.
            # So +90 is clockwise/East, + 180 is clockwise and -90 is anticlockwise/West for the wheel. But in teleopy
controller I switched east and west
            # So I need a good conversion
```

```
            self.publish_twist(0,0,0,0)
            rospy.sleep(5*self.transform_rate)
            self.publish_angle_bindings(False, 0)
            self.publish_feedback()

            translation = False

            if self.get_magnitude(distance_offset_vector) > 0.05:
                    translation = True

                    current_orientation_angle = self.get_z_angle(self.current_pose)
                    # This is in radians. This is angle of end effector w.r.t world. This is in radians. This will always
change as the robot moves.
                    print("\n--- Current Orientation Z-angle is ", math.degrees(current_orientation_angle))

                    distance_vector_angle = math.atan2(distance_offset_vector[1],distance_offset_vector[0])
                    # This is angle subtended by distance vector w.r.t world. This is in radians. This will always be
absolute.
                    print("\n--- Distance Vector angle is ", math.degrees(distance_vector_angle))

                    computed_angle = distance_vector_angle - current_orientation_angle
                    if computed_angle > math.pi:
                            computed_angle -= 2*math.pi
                    self.publish_angle_bindings(False, computed_angle)
                    self.publish_feedback()
                    print("\n--- Computed angle is ", math.degrees(computed_angle), " and distance offset is ",
self.get_magnitude(distance_offset_vector))


            while self.flag == 'Not OK': # Wait for wheels to turn
                    self.publish_feedback()

            factor = 0.1

            while translation is True:

                    turn = self.max_turn
                    speed = factor*self.max_speed

                    (vel, rot) = (1, 0)
                    vel = vel*speed
                    rot = rot*turn

                    self.publish_twist(speed, turn, vel, rot)
                    rospy.sleep(self.transform_rate)

                    posdiff = self.position_difference_vector(self.current_pose, self.last_pose)
                    self.linvel = (posdiff[0]/self.transform_rate, posdiff[1]/self.transform_rate, 0) # This is in meters
per second
                    distance_offset_vector = self.position_difference_vector(targetpose, self.current_pose)
                    distance_left = self.get_magnitude(distance_offset_vector)
                    print("\nDistance left to travel : ",distance_left)


                    current_orientation_angle = self.get_z_angle(self.current_pose)
                            # This is in radians. This is angle of end effector w.r.t world
                    distance_vector_angle = math.atan2(distance_offset_vector[1],distance_offset_vector[0])
                    # This is angle subtended by distance vector w.r.t world. This is in radians

                    new_computed_angle = distance_vector_angle - current_orientation_angle
                    if new_computed_angle > math.pi:
                            new_computed_angle -= 2*math.pi
                    elif new_computed_angle < -math.pi:
                            new_computed_angle += 2*math.pi
                    if abs(new_computed_angle - computed_angle) > math.radians(1):
                            print("\n--- New computed angle is ", math.degrees(new_computed_angle), " and distance
offset is ", distance_left)
                            computed_angle = new_computed_angle
                            self.publish_angle_bindings(False, new_computed_angle)

                    if distance_left <= 3 and distance_left > 0.5: # Apply Brakes
                            #factor = 0.5*distance_left
                            factor = 1
                    elif distance_left <= 0.5:
                            #factor = 0.25
                            factor = 0.5
                    else:
```

```
                        factor = 5

                self.publish_feedback()
                if distance_left < 0.05:
                        translation = False



        self.publish_twist(0,0,0,0)
        rospy.sleep(5*self.transform_rate)
        self.publish_angle_bindings(False, 0)
        self.publish_feedback()
        print("\n--- Translation complete")
        pass

    def execute(self, goal):
        print("Executing callback for AnkurbotController")

        if goal.action == "CALIBRATION":
                self.result.flag = True
                self.result.message = "SPACE=XY/AREA=17,7,15"
                self.server.set_succeeded(self.result)
        else:
                targetpose = goal.target
                print("\ntargetpose received by ankurbot : ",targetpose)

                if goal.action == "TRUE" or goal.action == "true" or goal.action == "True":
                        rotationflag = True # Rotate about centroid
                else:
                        rotationflag = False # Rotate about end-effector

                while self.last_pose is None: # Wait for at least one historical pose to be registered
                        pass

                # First compute if there is a need for rotation
                angle_offset = self.angle_difference(self.get_z_angle(targetpose),
self.get_z_angle(self.current_pose)) #This is in radians
                self.rotation_loop(angle_offset, targetpose, rotationflag)

                # Then compute if there is a need for translation
                distance_offset = self.position_difference_vector(targetpose, self.current_pose)
                self.translation_loop(targetpose, distance_offset)

                # Then compute if there is a need for angular correction
                angle_offset = self.angle_difference(self.get_z_angle(targetpose),
self.get_z_angle(self.current_pose)) #This is in radians
                self.rotation_loop(angle_offset, targetpose, rotationflag)

                self.result.flag = True
                self.result.message = ""
                self.server.set_succeeded(self.result)

class HCController:
    def __init__(self, frame):
        self.switch_pub = rospy.Publisher(frame + "/motion_start_stop", Int8, queue_size=5)
        self.state_sub = rospy.Subscriber(frame + "/state", Int8, self.update_state)
        self.joystick_pub = rospy.Publisher(frame + "/joy", Joy, queue_size=5)
        self.server = actionlib.SimpleActionServer(frame + '/move_robot',MoveRobotAction, self.execute, False)
        self.feedback = MoveRobotFeedback()
        self.result = MoveRobotResult()
        self.frame = frame

        self.state = 0
        self.current_pose = None
        self.last_pose = None
        self.transform_rate = 1

        self.server.start()

    def update_state(self, data):
        self.state = data.data
        print("\n=== For HC10, currently the state is ", self.state)

    def angle_difference(self, angle1, angle2):
        angle = (angle1 - angle2)%(2*math.pi)
        if angle > math.pi:
                angle -= 2*math.pi
```

```
        return angle

    def get_z_angle(self, pose):
        if isinstance(pose, TransformStamped):
                return euler_from_quaternion((pose.transform.rotation.x, pose.transform.rotation.y,
pose.transform.rotation.z, pose.transform.rotation.w))[2]
        elif isinstance(pose, Pose):
                return euler_from_quaternion((pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w))[2]

    def get_y_angle(self, pose):
        if isinstance(pose, TransformStamped):
                return euler_from_quaternion((pose.transform.rotation.x, pose.transform.rotation.y,
pose.transform.rotation.z, pose.transform.rotation.w))[1]
        elif isinstance(pose, Pose):
                return euler_from_quaternion((pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w))[1]

    def get_x_angle(self, pose):
        if isinstance(pose, TransformStamped):
                return euler_from_quaternion((pose.transform.rotation.x, pose.transform.rotation.y,
pose.transform.rotation.z, pose.transform.rotation.w))[0]
        elif isinstance(pose, Pose):
                return euler_from_quaternion((pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w))[0]

    def position_difference_vector(self, targetpose, initpose):
        if isinstance(initpose, TransformStamped):
                (xi, yi, zi) = (initpose.transform.translation.x, initpose.transform.translation.y,
initpose.transform.translation.z)
        elif isinstance(initpose, Pose):
                (xi, yi, zi) = (initpose.position.x, initpose.position.y, initpose.position.z)
        if isinstance(targetpose, TransformStamped):
                (xt, yt, zt) = (targetpose.transform.translation.x, targetpose.transform.translation.y,
targetpose.transform.translation.z)
        elif isinstance(targetpose, Pose):
                (xt, yt, zt) = (targetpose.position.x, targetpose.position.y, targetpose.position.z)
        return (xi - xt, yi - yt, zi - zt) # REVIEW THIS

    def set_current_pose(self, pose):
        if self.current_pose is None:
                self.current_pose = TransformStamped()
                self.current_pose = pose
        else:
                if self.last_pose is None:
                        self.last_pose = TransformStamped()
                        self.last_pose = self.current_pose
                else:
                        if rospy.Time(self.last_pose.header.stamp.secs,
self.last_pose.header.stamp.nsecs)!=rospy.Time(self.current_pose.header.stamp.secs,
self.current_pose.header.stamp.nsecs):
                                self.transform_rate = rospy.Time(self.current_pose.header.stamp.secs,
self.current_pose.header.stamp.nsecs).to_sec() - rospy.Time(self.last_pose.header.stamp.secs,
self.last_pose.header.stamp.nsecs).to_sec()
                        self.last_pose = self.current_pose
                        self.current_pose = pose

    def get_current_pose(self):
        return self.current_pose

    def publish_feedback(self):
        (self.feedback.vx, self.feedback.vy, self.feedback.vz) = self.position_difference_vector(self.current_pose,
self.last_pose)
        self.feedback.ax = self.angle_difference(self.get_x_angle(self.last_pose), self.get_x_angle(self.current_pose))
        self.feedback.ay = self.angle_difference(self.get_y_angle(self.last_pose), self.get_y_angle(self.current_pose))
        self.feedback.az = self.angle_difference(self.get_z_angle(self.last_pose), self.get_z_angle(self.current_pose))
        self.feedback.time = self.transform_rate
        self.feedback.robotname = self.frame
        self.feedback.robottype = "Manipulator"
        self.server.publish_feedback(self.feedback)

    def execute(self, goal):
        print("Executing callback for HC10 Controller")
        self.result = MoveRobotResult()
        if goal.action == "CALIBRATION":
                self.result.flag = True
                self.result.message = "SPACE=SWITCH/AREA=11,29,17,32,0,10"
```

```
                joycommand = Joy()
                joycommand.buttons = [0,0,1,0]
                joycommand.axes = [0,0,0,0]
                self.joystick_pub.publish(joycommand)
                while self.state != 3:
                        pass
                joycommand = Joy()
                joycommand.buttons = [0,0,0,1]
                joycommand.axes = [0,0,0,0]
                self.joystick_pub.publish(joycommand) # Publish Square Joystick command


        elif goal.action == "ON" or goal.action == "OFF":
                while self.state == 4:
                        if self.server.is_preempt_requested():
                                break
                        if goal.action == "ON":
                                self.switch_pub.publish(1)
                        elif goal.action == "OFF":
                                self.switch_pub.publish(0)
                        self.publish_feedback()
                self.result.flag = True
                self.result.message = ""

        elif goal.action == "CLEANUP":
                joycommand = Joy()
                joycommand.buttons = [0,0,0,0]
                joycommand.axes = [0,0,0,0]
                self.joystick_pub.publish(joycommand) # Publish Square Joystick command
                self.result.flag = True
                self.result.message = ""

        self.server.set_succeeded(self.result)


class MultiRobotControlsServer:
    def __init__(self):
        self.seeds_sub = rospy.Subscriber("seed_transforms", TransformStampedVector, self.update_seeds)
        self.database = {} # Maintain a list of all Robot Names as well as their current transforms and ActionServers

    def update_seeds(self, data):
        for idx in range(len(data.transforms)):
                if data.object[idx] in ["Mobile", "Manipulator"]:
                        frame = '/' + data.frame[idx].split('/')[0]
                        if frame not in self.database.keys():
                                if frame[0:9] == "/ankurbot":
                                        controller = AnkurbotController(frame)
                                        controller.set_current_pose(data.transforms[idx])
                                        self.database[frame] = controller
                                elif frame[0:10] == "/hc10_barc":
                                        controller = HCController(frame)
                                        controller.set_current_pose(data.transforms[idx])
                                        self.database[frame] = controller
                        else:
                                if frame[0:9] == "/ankurbot":
                                        self.database[frame].set_current_pose(data.transforms[idx])
                                elif frame[0:10] == "/hc10_barc":
                                        self.database[frame].set_current_pose(data.transforms[idx])


def main(args):
        mrcs = MultiRobotControlsServer()
        rospy.init_node('multi_robot_controls_server_node', anonymous=True)
        try:
                rospy.spin()
        except KeyboardInterrupt:
                print("Shutting down!")

if __name__ == '__main__':
    main(sys.argv)
```