**METE4300U Mobile Robotics**
**Milestone Report 4**
**Group 13**

| # | Last Name | First Name | Student ID |
|---|---|---|---|
| **1** | Elvin | Bryce | 100696601 |
| **2** | Slade | Cameron | 100703005 |
| **3** | Sprung | Lucas | 100699915 |

## Table of Contents

**Introduction**

The goal of Milestone 4 is to have the robot autonomously explore an unknown environment, and once it locates one of the marked packages, retrieve it and return it to its starting position. The robot must also avoid four obstacles that are placed in unknown locations in the mappable area. The mapping can be achieved utilizing Simultaneous Localization and Mapping (SLAM). Using SLAM, a LiDAR sensor can be used by the robot to create a 2-dimensional map of the environment, which the robot can then use to traverse the area. Identification of the packages can be achieved using a camera placed on the front of the robot which allows it to be located and then placed on the generated map.

Whether the area is known or not, the robot will have to create a path to its desired destination when traversing the region. While following the preferred route to a location, the robot must detect changes to the environment, use these discoveries, and avoid any obstacles that may impede the path.

Finally, the robot must save a location on the map and return to that location once exploration is complete. In the case of this milestone, the location is the starting location of the exploration. Upon returning to this location, the robot should stop to indicate that it has finished its task.

**Milestone Engineering Requirements**

This project is completed using a Turtlebot3 WafflePi mobile robot. This robot is approximately 20cmx20cm in length/width. This robot will be traversing and mapping an area of approximately 15 square meters. There is no time limit on this milestone task. Therefore, there is no required minimum speed the robot needs to move at, as long as the task is completed within a reasonable amount of time. The default speed the Turtlebot moves at is approximately 0.22m/s, which will be sufficient to complete the milestone.

In Milestone 1, the robot needs to create a full 2-dimensional map of the area using the Lidar sensor mounted on the robot's top. This map will need to update semi-frequently to confirm that the program is working. Updating the map approximately every 0.5 seconds provides a good balance between providing the updated information and not taking up too much time during the program. The robot also needs to save the information on its starting position to return to that point upon completion of the 2D map.

In addition to creating a 2-dimensional map of the 15 square meter area, for Milestone 2 the robot must locate two packages within the area using the front facing camera and indicate the location of each package on the map before returning to the starting position. The packages have a set dimension of 10cmx5cmx5cm and can be made out of any material. The packages may also be marked as it is seen fit with an Aruco or QR code for example. Obtaining the location of the packages along with the mapping of the area must be completed autonomously by the robot.

Milestone 3 as an additional requirement onto Milestone 2. While the robot is mapping, it must also be able to manouver around four obstacles located inside the mappable area. Two of

the obstacles must be of the dimension 30cmx20cmx20cm and the other two obstacles must be 20cmx15cmx15cm. The obstacles must be plain and not have any identifying markers.

Milestone 4 requirements don't build up on the previous milestones requirements but rather give a different objective using what was learnt from the previous milestones. For this milestone, the robot is required to locate one of the packages in the 15 square meter area, pick it up, and return home. The package must be picked up by the robot such that it is not dragging on the ground while it is being retrieved. The same four obstacles from Milestone 3 will be in the mappable area and must be avoided by the robot as well.

**Background Research**

In order to get a better idea of how this milestone in the project could be completed, some background research into current state of the art concepts of the milestone was conducted. The main technologies used in milestone 2 were LiDAR, SLAM and a front-facing camera with other concepts building off of these. Light Detection and Ranging (LiDAR) can be used to achieve a high resolution map of an indoor environment and the position of the device in the environment [1]. Simultaneous Localization and Mapping (SLAM) allows the robot to take its current position in a generated map and uses it to create and follow a planned path to a desired goal position [2]. Gmapping which at its core is a combination of LiDAR and SLAM would be able to use the main concepts of LiDAR and SLAM and combine them to a system that can both map an area of unknown size and use this map to complete a set goal. The front-facing camera that comes with the Turtlebot gives the sufficient means of locating specific markings on the packages that could be detected using the appropriate ROS package.

Background research also involved looking into different ROS packages and seeing how they could be used with our robot to execute the milestone. One of the main packages that was looked at was explore_lite which has seen much success with this kind of project in the past. The explore_lite package uses greedy frontier exploration to explore the environment the robot is in. It performs this until no area of the map or frontier can be found. The movement commands are then sent to the node called move_base. The explore_lite package does not create its own map which allows it to be more efficient and easier to control. It subscribes to messages from nav_msgs/OccupancyGrid and nav_msgs/OccupancyGridUpdate which are used to construct the map [3]. Frontier exploration was also looked into as it would also provide us with a map of the area but this was not used as it requires a defined boundary to explore whereas the greedy frontier exploration in explore_lite does not require bounds which is favoured in order to keep the robot fully autonomous. An additional benefit to explore_lite is the move_base which allows us to define a goal for the robot to achieve after the map is completed such as the need for the robot to the home position. A potential downside to this package is with no bound on the exploration, exploration of a specified area may take longer compared to a bounded exploration.

Another exploration package that was researched was rrt_exploration. This is a ROS package that is based on the Rapidly-Exploring Random Tree (RRT) algorithm which implements multi-robot map exploration [4] but can also be implemented for single robot

exploration. Similar to explore_lite, this package uses LiDAR to generate a map of its surroundings. One of the key differences is this ROS package uses frontier exploration which specifies points to go to for exploration compared to greedy frontier exploration which does not require these points. The rrt_exploration package apart from this operates similarly to explore_lite in the way it is not generating the map itself but is subscribing to nodes such as nav_msgs/OccupancyGrid to assist in building a map. This package would have great use for multi-robot mapping operations however the project only uses one robot and the fact of frontier exploration requiring specified points for map exploration could cause some limitations.

Early on it was determined that to aid in detection of the packages that they would have a marking in the form of either a QRcode or an aruco code. One package that was researched was visp_auto_tracker which is an automated pattern-based object tracker [5]. The object to be detected should have a QRcode or Flashcode pattern and based on this pattern, an object can be automatically detected. This package uses the image provided by the camera and once it detects the code can obtain its pose with respect to the camera. This package would be able to give the location of the package with a good degree of accuracy. Another package that was researched was aruco_detect which is also a fiducial detection package using the aruco library [6]. Like the visp_auto_tracker, aruco_detect uses information from a camera to obtain the position of a pattern which in this case is an aruco code. One of the differences between these packages is visp_auto_tracker uses a raw image for its image processing while aruco detect uses a compressed image. The vis_auto_tracker package offers a more versatile means of obtaining an object's location using multiple symbols; however, the package requires more setup in comparison to aruco detect. The aruco_detect package is more straightforward to set up but requires more transforms compared to visp_auto_tracker to obtain the position of an object relative to the map.

One requirement of the project that is not explicitly stated but is necessary is for the robot to be able to avoid the packages that are plotted on the map. This can be achieved in one of two ways by either using point clouds of fake LiDAR scans. Both solutions would yield the desired result but the implementation of each solution is different. Using the fiducial code, a point cloud could be placed on the package which would then place it on the cost map so the robot can see the packages rough location and avoid it. The other solution is to have a fake LiDAR scan be placed on the map when the fiducial is detected. Both solutions would be just as effective as the other but upon further research, the point cloud method was determined to be much easier to implement compared to faking LiDAR scans.

With the robot needing to pick up the obstacle and bring it home for the fourth milestone, planning was required to create a system that would accomplish this task as the turtlebot does not come with a means to do so. The milestone does not require that package to be put down after returning home which was a factor for deciding what method to use. One method considered was a spatula design which would slide under the package to pick it up. This method would work in theory but in application this would be unreliable as once the package is retrieved, there would not be much means of keeping the package from falling off and the spatula would need to be

raised off the ground for the design to meet the requirements of the project. Another design considered was a servo gripper that would grab the package and lift it off the ground. This design would be the most ideal for the milestone as it would be able to grab the package securely but this solution would be the most complex solution to implement and might not be feasible given the time to obtain all the necessary parts for the design. One other design looked to was the use of magnets. Magnets could be used on the front of the robot as well as on the package to allow the robot to pick up the package. The benefit of this system would be that the package would be secured given that the magnets are strong enough and would be easy to implement. A potential downside would be that the package would not be removable without additional means but since it is not a requirement to drop off the package, this is still a usable solution for the milestone. One consideration would also be to combine some of these concepts such as magnets and a servo arm which would do a good job at obtaining and securing the package.

## Concept Generation

### Package Grab Mechanism

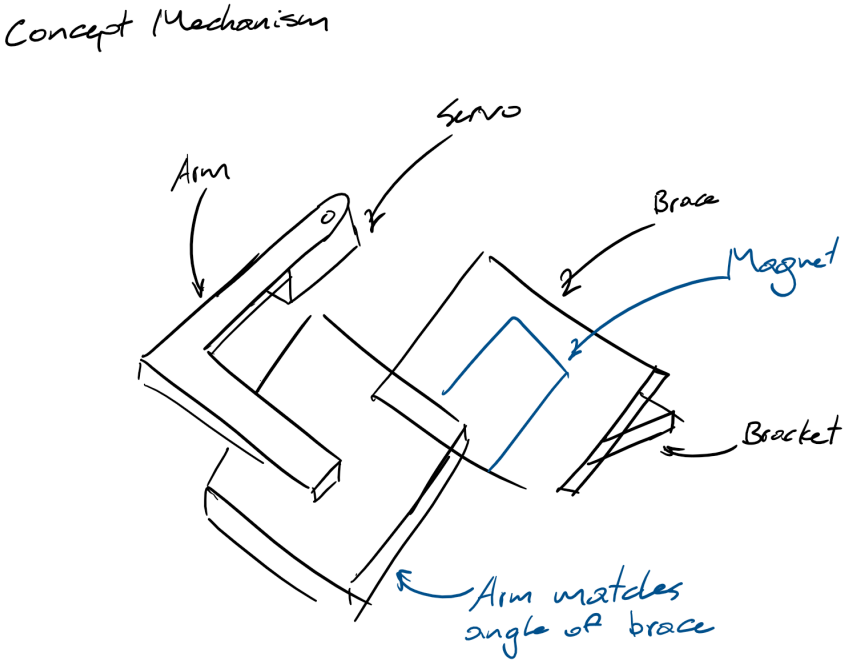
**House of Quality**
Mobile Robotics

| | Customer Requirements | Engineering Requirements: Time to locate package (<5min) | Package detection distance (>2m) | Payload size (>200g) | SLAM update frequency (<1.0s) | System response time (<0.4s) | Concept Comparison: Suction cup gripper | Actuated arm pickup | Magnet bar | Magnet with servo arm | Suction cup with servo arm | Forklift method |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Pick up a package | | ◆ | ★ | ◆ | ◆ | 4 | 3 | 4 | 5 | 5 | 3 |
| 2 | Return the package to starting position | | | O | ◆ | | 5 | 4 | 5 | 5 | 5 | 2 |
| 3 | Protect package from damage | | | ★ | | O | 4 | 3 | 3 | 4 | 4 | 1 |
| 4 | Locate package autonomously (good visibility) | ★ | O | | ◆ | ★ | 5 | 2 | 3 | 4 | 4 | 5 |
| 5 | Navigate unknown environment (small size) | | ◆ | O | | | 3 | 2 | 3 | 4 | 2 | 3 |
| | Weight | 40 | 70 | 90 | 30 | 60 | 21 | 14 | 18 | 22 | 20 | 14 |
| | Percent | 14% | 24% | 31% | 10% | 21% | | | | | | |

| LEGEND | |
|---|---|
| ◆ | Weak relationship |
| O | Medium relationship |
| ★ | Strong relationship |

According to the engineering specifications, the package cannot drag on the ground, and must be lifted, even slightly, off the ground. Initial thoughts were to use a servo to both grab the package, then lift it off the ground. This however would require more than one motor, and it was desired to use as little additional components as possible. The project outline states that the packages could be made of any material, so thought was then given to the possibility of using magnets in the system. If a magnet could be placed on the robot, then a reciprocating magnet could be placed inside the packages to allow them to be picked up. There were however issues with not being able to drag the package around, then needs to be off the ground. A concept was

then created where a brace was attached to the front of the robot at an incline, so that when a package was magnetized to the brace, it would tip backwards and lift off the ground. During this concept generation it was also a desired idea to use a servo arm to assist the magnets on the brace, so that if the robot did not approach the package at a head on angle, the arm could direct the package to the front of the robot where the magnet was. The servo arm could have a large paddle that matches the angle of the brace so that when the arm helps to hold the package to the robot, it can also help maintain the tipping angle. A sketch of the conceptualized system can be seen below




**Functional Decomposition of Developed System**

**Milestone One**

The program used for the first milestone makes use of the explore_lite ROS package. Other packages used in the program for the milestone are a part of the default turtlebot kit. Frontier exploration is specifically designed for situations exactly like this one, where an unknown area must be mapped. In this case, explore_lite was used since it implements greedy frontier exploration, allowing it to search an unbounded area until it can find no more frontiers to explore. A frontier is simply the border between a known area and an unknown area.

The gmapping ROS package uses the LiDAR data from the robot to create this map, and the explore_lite package requires a map to find new frontiers.

Finally, in order to actually pathfind to a new location when a request is delivered, the navigation package for ROS was used. This package takes a goal configuration, and attempts to find a path to that location avoiding all obstacles.

Modifications were made to the source file of the package in order to perform the required tasks. The edits made to the source file were as follows: A line was written at the beginning of the file that saves the current position information on the robot so that it knows

what location and orientation it needs to return to upon completion of the map. At the end of the mapping, the message that displays “exploration complete” was changed to send a goal to the move base telling the robot to return to the location and orientation that was saved at the beginning of the program.

A full documentation of the modifications to the code can be found below. For the full source code of the modified explore_lite package, please refer to the Appendix. Firstly, a new function was implemented that simply takes some set of odometry data, and if the program has yet to save an end goal for the system, it saves the odometry data as the end goal.

```
// gotStart variable to store if a start position is saved yet
bool gotStart = false;

void setPosition(const nav_msgs::Odometry::ConstPtr& msg) {
  // Do not execute the function if a starting position has already been saved
  if (gotStart) return;
  gotStart = true;
  ROS_INFO("Attempting to save position");
  // Set the start goal using the odometry data from the subscribed topic
  startPositionGoal.target_pose.header.frame_id = "map";
  startPositionGoal.target_pose.header.stamp = ros::Time::now();
  startPositionGoal.target_pose.pose.position.x = msg->pose.pose.position.x;
  startPositionGoal.target_pose.pose.position.y = msg->pose.pose.position.y;
  startPositionGoal.target_pose.pose.orientation.w = 1.0;
}
```

After adding this new function, it had to be made a callback function for the odometry data if one wants the starting location. This can be done in a single line as shown below.

```
  // subscriber to save start position using odom topic
  posSub = relative_nh_.subscribe("odom", 1000, setPosition);
  ROS_INFO("Subscribing to Odom data");
```

Then finally, once exploration has finished, a few lines were added to send the saved start position as a goal to the move_base topic.

```
void Explore::stop() {
  move_base_client_.cancelAllGoals();
  exploring_timer_.stop();
  ROS_INFO("Exploration stopped.");
  // after exploration stops, return to start position
  move_base_client_.sendGoal(startPositionGoal);
  move_base_client_.waitForResult();
  ROS_INFO("Successfully returned home.");
}
```

move_base is the topic that tells the robot to go to whichever position is given to it. A launch file was created to run the explore light and move base commands in a single file. This

way the robot can perform all tasks from start to finish without any human interference. The UML diagram shown in figure () shows the major steps of the program from start to finish.

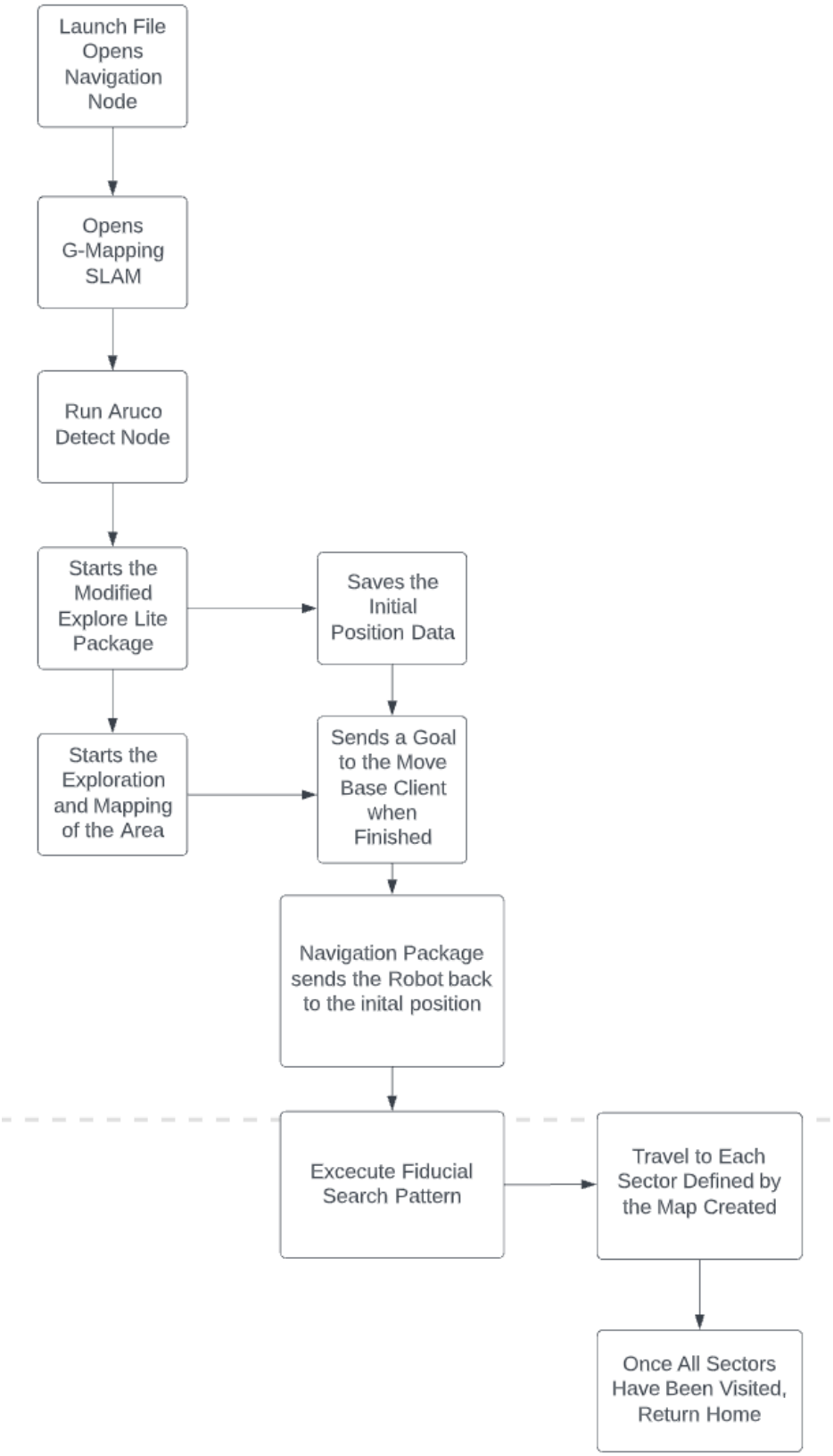

Launch File Opens Navigation Node
Opens G-Mapping SLAM
Run Aruco Detect Node
Starts the Modified Explore Lite Package
Saves the Initial Position Data
Starts the Exploration and Mapping of the Area
Sends a Goal to the Move Base Client when Finished
Navigation Package sends the Robot back to the inital position
Excecute Fiducial Search Pattern
Travel to Each Sector Defined by the Map Created
Once All Sectors Have Been Visited, Return Home

*Figure 1*

**Milestone Two**

For Milestone Two, the robot must be able to detect packages in the explorable area. Since Aruco markers are being used for this phase, they were printed to scale to be 5cm wide and are placed in the area. Since relying entirely on the exploration to detect the packages using a front-facing camera is unlikely, there also needed to be some means of searching the environment. A search pattern had to be designed and implemented.

Firstly, the most simple task, the robot used the aruco_detect library to detect Aruco markers using the front-facing camera. This library does a great job of detecting the markers, but it is not capable of marking them on RViz, which is a requirement for the milestone. A package was created relating to these "fiducial" markers to search and process the information from aruco_detect. The first node created simply accessed the transfer frames of the map and the fiducials and used the location information to plot them on RViz using a Marker message. The robot can only use this node to mark a single fiducial at a time due to limitations, and so it allows an argument to be passed to specify which specific fiducial this node is marking on the map.

```
std::string param;
int code_num;
_nh.getParam("fiducial_code", param);
code_num = atoi(param.substr(9, 12).c_str());
ROS_INFO(param.c_str());
```

It later publishes the marker for RViz to use, using a simple C++ publisher setup for ROS. This marker can only be seen if manually targeted in RViz; however, it can be seen during and long after program execution.

```
  geometry_msgs::TransformStamped transform;
  try {
    transform = tfBuffer.lookupTransform("map", param.c_str(), ros::Time(0));
  } catch (tf2::TransformException &ex) {
    ros::Duration(0.5).sleep();
    continue;
  }
  marker.pose.position.x = transform.transform.translation.x;
  marker.pose.position.y = transform.transform.translation.y;
  marker.pose.position.z = transform.transform.translation.z;
  vis_pub.publish(marker);
```

This publisher is run continuously to allow it to keep updating the positions of the markers if need be.

Next, a search pattern had to be designed. Since this is a simple task, a simple search pattern algorithm was made. Simply, the node shall retrieve the Occupancy Grid from the SLAM node and load it as a map of the traversable area. Then, the node will split the map into 0.8m² cells. The node can then determine which ones are safely traversable and travel to each cell. This pattern will give a much higher chance of finding the packages in the camera lens. The robot is

programmed to face the same arbitrary direction at each cell so that there are little inconsistencies related to unideal orientation patterns. Although tested in multiple simulated environments, a single example of this process visualized can be seen below.

*Figure 2*

The illustration above explains the logic that the search pattern will follow. Firstly, the red lines indicate the bounds of the known region. Next, the region is split into cells, and each cell is evaluated to determine if the robot can fit within the cell safely (without risk of collision with a wall). If the cell is deemed safe, a green dot is placed there, representing a goal that the node will eventually send to the move_base client. The red dot is the map's origin according to the Occupancy Grid header information provided with the message.

This node was programmed entirely in Python for fast prototyping. Since the Occupancy Grid is provided as a grid and not a continuous system, there must be a way to convert a coordinate from this discrete frame to a real, continuous frame for the move_base client to use as a goal. Using the resolution of the grid and the origin information, a function was made to convert from these two coordinate systems.

```
def xy_to_m2(x, y):
    width    = map_data.info.width
    height   = map_data.info.height
    x_origin = width  / 2 + map_data.info.origin.position.x
```

```
    y_origin = height / 2 + map_data.info.origin.position.y
    new_x    = map_data.info.resolution * (x - x_origin)
    new_y    = map_data.info.resolution * (y - y_origin)
    return (new_x, new_y)
```

After this, a function was made to define a home point, similar as to what was done for Milestone One in the explore_lite package modifications.

```
def save_home_odom(msg):
    global home_goal
    if home_goal is not None:
        return
    home_goal = MoveBaseGoal()
    home_goal.target_pose.header.frame_id    = "map"
    home_goal.target_pose.pose.position.x    = msg.pose.pose.position.x
    home_goal.target_pose.pose.position.y    = msg.pose.pose.position.y
    home_goal.target_pose.pose.orientation.w = msg.pose.pose.orientation.w
```

Since the Occupancy Grid is stored in a single row array to represent a 2-dimensional space, a function was also made to convert from an x-y coordinate to an index in the array.

```
def get_index(x, y, scale):
    return x + y * scale
```

In order to validate a location, a box search around the location will be done, verifying a certain number of resolution pixels away from any obstacles or unknown space. A function was made to fulfill this task.

```
def validate_location(x, y, scale, dist_from_walls):
    valid = True
    for x_index in range(x - dist_from_walls, x + dist_from_walls):
        for y_index in range(y - dist_from_walls, y + dist_from_walls):
            p = map_data.data[get_index(x_index, y_index, scale)]
            if p != 0:
                valid = False
                break
        if not valid:
            break
    return valid
```

Now a callback function was made for both the Occupancy Grid and the Exploration nodes.

```
def exploration_finished_callback(msg):
    global map_data
    map_data = msg
    rospy.loginfo("Exploration complete. Beginning search.")
```

```
    search_map(map_data)

def map_data_callback(data):
    global map_data
    map_data = data
```

All of these subscribers and clients are activated and used within the main function of the program as shown below.

```
def main_node():
    map_sub = rospy.Subscriber('map', OccupancyGrid, map_data_callback)
    exp_sub = rospy.Subscriber(
        'exploration_msgs',
        String,
        exploration_finished_callback
    )
    hom_sub = rospy.Subscriber('odom', Odometry, save_home_odom)
    rospy.init_node('fiducial_search_node', anonymous=True)
    global move_base_client
    move_base_client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
    rospy.loginfo("Search node started. Waiting for move_base client...")
    move_base_client.wait_for_server()
    rospy.loginfo("Client detected. Waiting for exploration to complete...")
    rospy.spin()
```

Now the actual search function can be designed. This will be broken into smaller pieces for ease of explanation, but a complete version of the code is in the appendix.

The node needs a definition of some select locations on the Occupancy Grid for the search to start. It needs the top and bottom rows of the search area, and the left and right columns. It will also need a sector size for splitting the area into cells. All of these variables are first defined.

```
def search_map(map):
    row_top     = 0
    row_bottom  = 0
    col_left    = 0
    col_right   = 0
    sector_size = 0.8
```

Next, the algorithm will need to search one by one through these variables to find their respective values. This is done by traversing the rows and columns and checking for the first and last instance of a known area. Below is the loop for detecting the top known row of the Occupancy Grid.

```
    rospy.loginfo("Detecting bounds of search...")
    found_wall  = False
    for i in range(0, map.info.height):
```

```
        for j in range(0, map.info.width):
            if (map.data[get_index(j, i, map.info.width)] != -1):
                row_top = i
                found_wall = True
                break
        if found_wall:
            break
```

For the sake of brevity, the other row and column searches will be left out of explanation, as they are very similar to the above.

Once the bounds of the search are located, the function will go through each row and column in the search area, by cell size increments. Since sector_size was defined in meters, it needs to be converted to cells. This is done inside the for loop. If the position is validated, then the x and y coordinates in real-space are determined and the goal is sent to the move_base client for the robot to navigate to that location. This is repeated until all traversable cells are exhausted. If a cell was determined traversable but no path can be found, it will be skipped upon wait_for_result()'s return call.

```
    rospy.loginfo("Search bounds located. Beginning sector sweep.")
    for col in range(col_left, col_right, sector_size / map.info.resolution):
        for row in range(row_top, row_bottom, sector_size / map.info.resolution):
            if validate_location(round(col), round(row), map.info.width, 3):
                x, y = xy_to_m2(col, row)
                rospy("Searching location (" + str(x) + ", " + str(y) + ")...")
                goal = MoveBaseGoal()
                goal.target_pose.header.frame_id    = "map"
                goal.target_pose.header.stamp       = rospy.Time.now()
                goal.target_pose.pose.position.x    = x
                goal.target_pose.pose.position.y    = y
                goal.target_pose.pose.orientation.w = 1.0
                move_base_client.send_goal(goal)
                move_base_client.wait_for_result()
```

Finally, the function will finish its search and return to the home point established in the odometry subscriber callback function.

```
    rospy.loginfo("Search complete. Returning home.")
    home_goal.target_pose.header.stamp = rospy.Time.now()
    move_base_client.send_goal(home_goal)
    move_base_client.wait_for_result()
```

This node is capable of the search pattern and is ready for its performance to be evaluated in a simulated environment.

**Milestone Three**

In Milestone Three, the additional main obstacle to overcome was the obstacles themselves. There was the addition of four more obstacles to the search area. Although detectable by the LiDAR, there is now an extra obstruction that can block the view of the package with the camera, making the search algorithm even more critical. In addition to new obstacles, there now needs to be a means of avoiding packages, so the robot does not damage them. This requirement means that there needs to be a means of conveying the package location information to the move_base system being used so that it may identify the packages as a keep-out zone.

The search algorithm was not changed for phase 3, as it functioned well in the natural and virtual environment during testing. However, a small change had to be made in order for packages to be avoided by the robot. Again in Python, a node was created that publishes the package's current location; however, this time, the node publishes a point cloud consisting of a circle around the package location on the map. This detail is vital because the move_base can read two primary types of sensor information: point clouds and laser scans. Point clouds are much easier to manipulate than laser scans, as they can be published to an arbitrary topic and then fed to the move_base as a new sensor type. This way, no editing of the cost map or any lower-level systems is required.

To begin with the node script, the global variables necessary for function are the inflation and the circle points parameters.

```
inflation = 0.3
circle_points = 25.0
```

The inflation parameter determines the diameter of the point cloud around the package location. This variable is essential for two reasons: the Aruco code does not represent the true center of the package, and the robot only avoids point clouds from contacting its center of mass. The circle points variable is the number of points to place around the package forming the circle. This number needs to be high enough that the robot decides to avoid the points altogether and not attempt to travel through them, but it should not be much higher than is necessary to reduce path planning time.

Firstly, to begin with, in the actual drawing of the circle, a simple function was created to return the points on a circle about a specific x and y coordinate. This function uses the diameter and number of points provided by the global variables.

```
def make_circle(x, y):
    cloud_points = []
    for i in range(0, int(circle_points)):
        cloud_points.append(
            [
                x + inflation / 2 * math.sin(2 * float(i) / circle_points * math.pi),
```

```
                y + inflation / 2 * math.cos(2 * float(i) / circle_points * math.pi),
                0.05
            ]
        )
    return cloud_points
```

The function above takes the location provided and uses basic trigonometry to find the unit circle around that point, then multiplies it by the radius to increase the size of the circle. An important thing to note in the function is the z-axis parameter of the cloud points. This parameter determines the height of the points off the ground and is very important for path planning. If this value is too high, the robot will not travel around the package and instead try to pass under it. However, too low, and the robot will think that it can safely travel over it. During testing, it was found that a height of 0.1m was too high and 0.0m too low. A safe z-axis value was 0.05m for consistent avoidance of the package.

Now to discuss the primary function of the node. The transform retrieval system is the same as the C++ transform system described previously, except it is now done in Python. For that reason, it will be displayed, but no further explanation will be provided.

```
def main():
    global pkg1
    global pkg2
    rospy.init_node('avoid_packages_node')
    pcl_pub = rospy.Publisher("/package_cloud", PointCloud2, queue_size=100)
    rospy.loginfo("Initializing package avoidance node...")
    rospy.sleep(1.0)

    tf_buffer = tf2_ros.Buffer()
    tf_listener = tf2_ros.TransformListener(tf_buffer)
```

The node is initialized above, and a publisher is defined to create the point clouds. The sleep is necessary here, as sometimes the node would have overlapping requests for transforms here, and this would cause it to crash.

The point cloud publisher requires a header providing the robot with a timestamp for the point cloud. This header allows it to determine which points are essential to note and avoid and which hold less weight on the path planning process.

```
    header = std_msgs.msg.Header()
    header.stamp = rospy.Time.now()
    header.frame_id = 'map'
```

The rest of the program takes place within a permanent while loop. The cloud points list is created, and the system will attempt to grab the transform for the first fiducial marker. If this fiducial marker does not yet have a transform (meaning it has not been located yet), it will throw an error. The exception statement will allow the node to continue functioning when this error is

thrown. The 'finally' statement allows it to attempt to retrieve the second package transform regardless of the state of the first package transform.

```
try:
    rospy.sleep(0.01)
    pkg1 = tf_buffer.lookup_transform("map", "fiducial_101", rospy.Time())
    rospy.loginfo("Transform 101 successfully located")
    cloud_points.append(make_circle(pkg1.transform.translation.x, pkg1.transform.translation.y))
except Exception as Argument:
    if pkg1 is not None:
        cloud_points.append(make_circle(pkg1.transform.translation.x, pkg1.transform.translation.y))
    else:
        rospy.loginfo("Fiducial 101 is not yet detected. " + str(Argument))
finally:
    try:
        rospy.sleep(0.01)
        pkg2 = tf_buffer.lookup_transform("map", "fiducial_100", rospy.Time())
        rospy.loginfo("Transform 100 successfully located")
        cloud_points.append(make_circle(pkg2.transform.translation.x, pkg2.transform.translation.y))
    except:
        if pkg2 is not None:
            cloud_points.append(
                make_circle(
                    pkg2.transform.translation.x,
                    Pkg2.transform.translation.y
                )
            )
        rospy.loginfo("Fiducial 100 is not yet detected. " + str(Argument))
```

Finally, so long as a package is defined, the point cloud is appended to the list of points. Now the point cloud is created by passing the list of points to the create_cloud_xyz32 method provided by the PointCloud2 Python module.

```
    pcl_pkg = pcl2.create_cloud_xyz32(
        header,
        cloud_points
    )
    pcl_pub.publish(pcl_pkg)
```

With the use of this node, the point cloud is published; however, complete avoidance is yet to be implemented. Although this is an easy change to make, the location of the change and the exact syntax of it is hard to locate. After some research, the parameters file in move_base needed to be edited to include a new definition of a sensor information topic and its description.

```
observation_sources: scan point_cloud
scan: {sensor_frame: base_scan, data_type: LaserScan, topic: scan, marking: true, clearing: true}
point_cloud: {sensor_frame: map, data_type: PointCloud2, topic: package_cloud, marking: true,
clearing: true}
```

**Milestone Four**

Milestone four required a significant increase in the overall complexity of the system. Since the chosen design requires the actuation of a servo motor, there needs to be some method to access the GPIO pins on the Raspberry Pi. There were a few different means of achieving this, but the chosen method was to create a service alongside a client and server to send commands to be run on the Raspberry Pi. Luckily, Python already contains built in libraries to control the GPIO pins of the Raspberry Pi that are pre-installed. Since Python can also operate in ROS, this was a perfect opportunity to create a server and a client.

The server side is run on the Raspberry Pi. This is because the server will receive requests from the client, and use the request information to set the output of the GPIO pins. Firstly, a format for the service had to be chosen. The service is defined as follows.

```
int64 pin
float64 value
---
string res
```

The service takes an integer value corresponding to the pin that the control should be applied to. The float value is the actual duty cycle to be sent to the pin. The duty cycle to achieve the desired angle will have to be tested and decided upon experimentally. Finally, the service returns a string response, which will just contain the status of the command. If this result returns "OK" it means that the pin was set successfully.

```
import rospy
from gpio_control.srv import SetGPIO
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
GPIO.setup(20, GPIO.OUT)
p = GPIO.PWM(20, 50)
```

The server begins above by importing the service that was created, as well as the Raspberry Pi GPIO module, and setting the basic parameters of the GPIO system. Since the pins need to be a PWM signal, certain pins will need to be set up. For this server, only pin 20 is set up to be a PWM signal, and the rest are considered digital outputs. The frequency is set to 50 Hz.

```
def set_pin_callback(req):
    rospy.loginfo(req)
    p.ChangeDutyCycle(req.value)
    return {
        'res': 'OK'
    }
```

The callback for the pin being set simply logs that it received a request and then sets the duty cycle on the GPIO pin and returns an OK reply assuming that no errors occurred during the operation.

```
def main():
    rospy.init_node('servo_service_server')
    rospy.Service('set_gpio_output', SetGPIO, set_pin_callback)
    rospy.loginfo('Server service started. Ready to receive requests.')
    rospy.spin()
    GPIO.cleanup()
```

The main function of the server just sets up the service for requests and keeps the system running to receive requests. This now leaves a solid framework for a client to connect to the system.

The client must be run alongside the server to send commands to the Raspberry Pi GPIO pins. The client will be run on the remote device, and simply needs to contact the server to send commands.

```
def control_gpio(msg):
    global set_gpio_output
    rospy.wait_for_service('set_gpio_output')
    rospy.loginfo('Received a message: ' + msg.data)
    if msg.data == "close_servo":
        try:
            for i in range(20):
                set_gpio_output(20, 2 + (i + 1) * 0.25)
                rospy.sleep(2.0 / 20.0)
        except rospy.ServiceException as e:
            rospy.logwarn(e)
    elif msg.data == "open_servo":
        try:
            set_gpio_output(20, 2)
        except rospy.ServiceException as e:
            rospy.logwarn(e)
```

First a callback function was created that takes a string as a message and sets the GPIO pins based on the result of that string. Currently, the function accepts two different inputs, “close_servo” and “open_servo”. The command to open the servo will set the PWM to a signal that immediately opens the servo motor completely. However, on closing, a slower control is desired to prevent the package being knocked completely out of the way of the arm during operation. For this, a simple for loop was used to increment the angle. These angles (the second argument in set_gpio_output) were determined via experimentation.

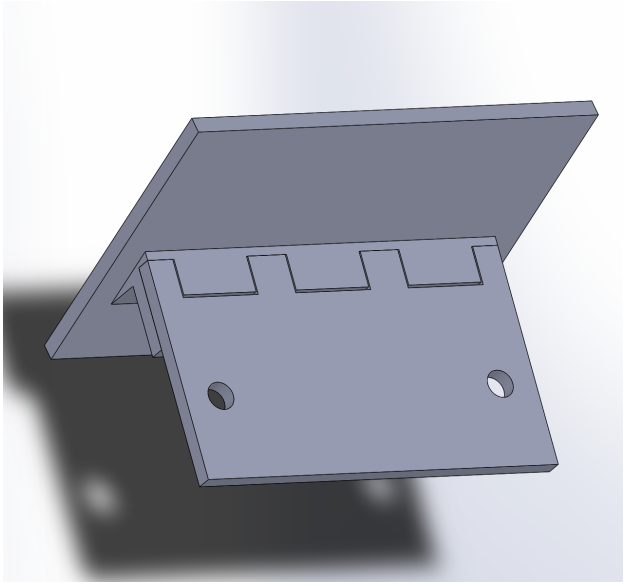
```
def main():
    rospy.init_node('servo_service_client_node')
    gpio_sub = rospy.Subscriber('gpio_msgs', String, control_gpio)
    global set_gpio_output
```

```
    set_gpio_output = rospy.ServiceProxy('set_gpio_output', SetGPIO)
    rospy.wait_for_service('set_gpio_output')
    set_gpio_output(20, 2)
    rospy.spin()
```

The main function of the client sets up a subscriber to receive the open and close commands, and then also sets up a service proxy to connect to the server and send GPIO commands to the Raspberry Pi itself. Since the server was kept separate from the client, and the client operates on the remote device, the server side would not need to be adjusted at all once implemented. All design focus can be placed on the client side of the operation, which is more accessible than the Raspberry Pi.

**Form Design and Engineering Analysis**

Phase 4 required a method to pick up the package and bring it back to the robots starting position. The concepts mentioned were modeled and manufactured to be able to help the robot with this additional requirement. The system worked by using an actuated arm to assist a magnet that holds the box against the robot. The arm was actuated with a small servo motor that is connected to the pins on the raspberry pi controller. The servo arm was attached to the turtlebot by manufacturing a bracket that holds the servo motor in a cage so that its fixed to the bot. The bracket has holes that are spaced to match holes on the turtlebot so that screws can be used to hold the bracket to the bot. Another bracket was made to hold a brace that the magnet is fixed to. The bracket uses similar holes spaced out for the holes on the bot, and then a toothed edge that faces outwards where the brace attaches. The brace is designed with the opposite teeth so that it can mesh with the teeth on the bracket. A visual of the teeth design for the bracket and brace can be seen below



The brace that holds the magnet for the package sits at a 15 degree angle from the vertical axis so that when the package is placed on the brace, it tips backwards, lifting it off the ground without the need for actuation on the brace. The arm attached to the servo that is used to help direct the package to the magnet has a matching incline so that when it holds the package to the brace it helps to keep the package tipped and off the ground.

**Design for Manufacturing**

All of the parts designed were 3D printed, and therefore designed with as minimal overhang as possible to limit the amount of support material needed. The holes for the screws, motor cage, and teeth on the bracket and brace were designed with the tolerances and errors of the machine in mind. The holes did not need to contain a thread because the screws use a nut to hold them in place, so the holes only need to be big enough that the screw can fit inside them without being so big that the head does not fit through the hole.

Other components necessary for this mechanism were purchased parts, including an SG90 servo motor and ceramic magnets. Gorilla tape was used to help hold the magnet to the brace, as well as keep the brace held to the bracket.

**Design for Safety**

When designing the autonomous system to be used in the robot many considerations had to be made towards the safety of operation. For example, the robot must obviously not damage itself during operation. The robot should be clear of any walls or obstacles, and avoid tipping any objects in the area for the safety of the robot itself. Besides the safety of the robot, it is also important to verify the safety of the packages as they are not to be damaged under any circumstances.

Firstly, the robot control system in place for the Turtlebot3 navigation module allows for the input of an “inflation” parameter, which inflates obstacles to be larger than they are in reality. This inflation allows the robot to avoid obstacles with a safety factor in mind, and keep a distance from the obstacles rather than taking the absolute shortest path. The inflation parameter on the robot initially was only 0.8m, but was increased to 1.0m to reduce the likelihood of a close approach, which can slow operation and even result in a standstill.

Next, the packages had to be accounted for. Besides avoiding the packages, which was a requirement for one of the milestones, the method to grab the package also had to be safe to the packages. It is preferable that the packages is dragged as little as possible, and necessary that it does not touch the ground during the time that the robot is carrying it. This limitation immediately eliminated some of the options for picking of the packages, as they were too risky (e.g. could knock the package over) or too damaging (i.e. impale the package). For this reason, the design was focused on minimal systems, such as magnets and suction cup designs. The magnet design was preferable as it not only allowed for a means of picking a package up, but it can also be placed higher on the robot to lift the package further off of the ground, ensuring that there is no dragging. Furthermore, for additional safety, an arm was added to the robot to prevent

the package from becoming dislodged during motion, and to ensure that the package is properly attached to the magnet, and not being dragged on the front.

**Test Plan/Results**

The test plan for Milestone Four was less intensive than in the earlier phases. There would still be obstacles, but they can be detected by the LiDAR. There is also the same number of packages, and the same size of room. The main difference besides the retrieval of the packages was the removal of the requirements to map the region and plot the location of the packages. This means that the testing plan for the milestone is focused mainly on the retrieval of packages. Now that the location of the packages is more important, the tolerances are much lower for acceptable error in estimated position; likely the transforms would have to be tested and tweaked once again. Furthermore, the means of retrieving the package had to be tested. Retrieving the package is difficult to test without the detection of the package, so this would not be completely possible until the previously discussed tests were completed.

Upon the testing of the location of the packages, the robot was commanded to travel to exactly the location that the packages were estimated to be at, and evaluate the distance from that position to the actual location of the package. In testing, the robot was usually within 5 cm of the actual location of the package, but this is not acceptable for the means of securing the package that were chosen. The package can be slightly to the left or right of the robot, but too far and the magnets and arm will fail to retrieve the package. After some adjustments to the transform and to the location traveled to, the consistency of this system was greatly improved.

Once the package location tracking was adjusted and determined to be accurate, the focus was set to retrieving the package. Luckily not much work was required in this department, as the magnets were more than capable of retrieving the package; in fact, the magnets were strong enough to allow for a decent amount of error in package location compared to what was expected.

There were a large amount of problems encountered during the testing of the robot, but it was unclear whether or not these problems were the result of the underlying code of the installed packages, or the packages that were created specifically for this project.

The primary problem with this milestone was an issue with the time synchronization between the Raspberry Pi and the remote computer. The remote computer was not receiving the information from the Pi quickly enough, and this would cause a failure of the navigation systems. Many things were tried to fix this problem. One possible solution was to increase the transform tolerance, which has no effect on the problem. Another solution was to completely reset the system, wipe the cache, and recompile the code; this also had no effect on the problem. Multiple different WiFi networks were attempted, and different packages and boot orders were attempted, but none of these fixes even improved the consistency of operation. The chosen method of operating the system was to repeatedly restart the system completely until it worked. Using this method, a handful of attempts were made in which the system did not reach a standstill and receive the information of the transforms in a reasonable amount of time.

After many trials, a satisfactory run of Milestone Four was completed. It is unclear if the issue lies with one of the following three items: internet connectivity, hardware limitations, or some other third thing.

**Conclusion**

The robot could complete the demonstration, but in testing, it was unreliable. It is unfortunate that the issues suffered during the milestone were mostly out of the control of the group, and did not prove to be a simple thing to fix. The only solution that may fix the system that was not attempted was to use a completely new Turtlebot system, as neither of the two laptops available were capable of avoiding this frequent error.

Moving forward, there are some improvements that could be made to the system. Firstly, the search algorithm is not as robust as it could be. A search package could be used instead of attempting to design one from scratch, and this would likely also allow for a higher accuracy in the locating of the packages. There are also many issues with the sector size, and feasibility analysis of selected points. These would already be solved problems on the majority of pre-made search algorithms.

Another thing that could have been done differently from the start was the use of the exploration module and the amount of modifications made. In hindsight, many of the modifications were hastily placed in places that were not meaningful, and much of the code could be refactored or replaced. This issue was because ROS was being learned as development was done. With the knowledge gained from the project, much could have been done better in this aspect.

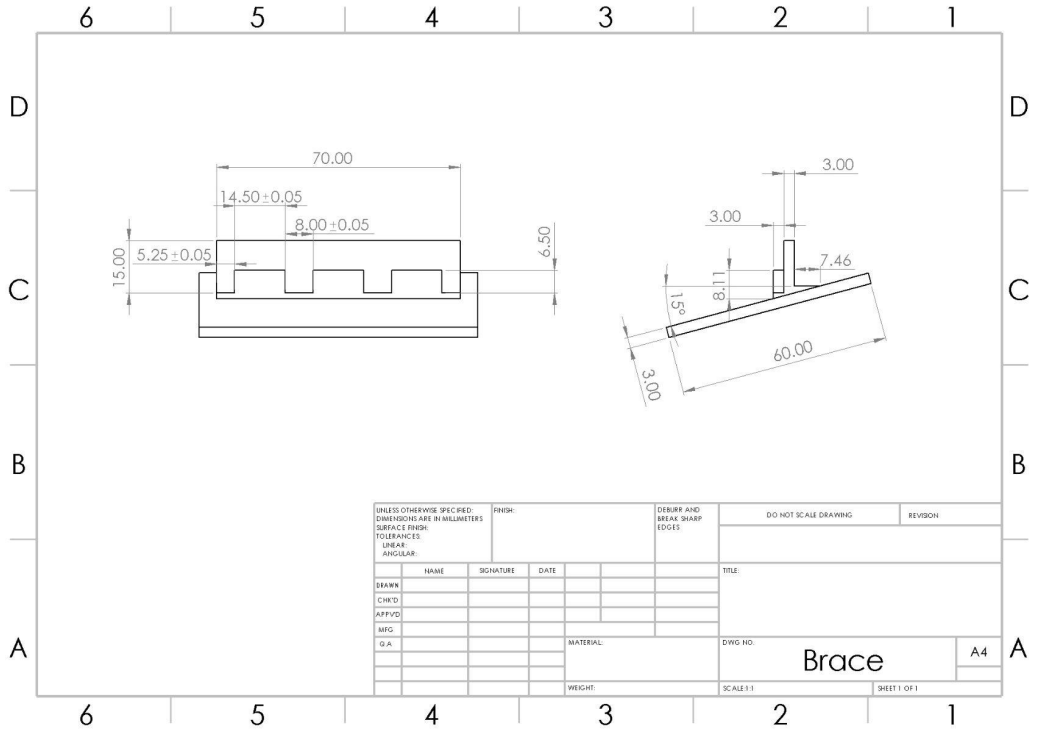
There were also some concerns with the power of the Raspberry Pi. Replacing the board with a stronger alternative single-board computer may have improved the connection issues, and also allowed for more on-board processing; this may eliminate some of the need for a more powerful remote computer.

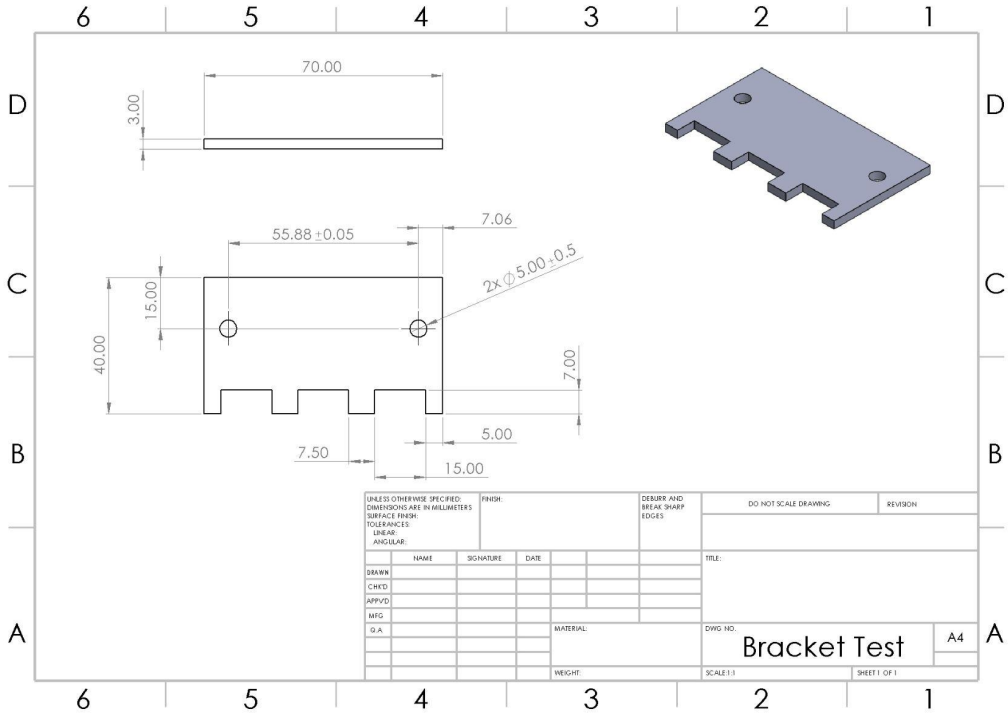
Finally, since all of the packages for the Turtlebot3 came pre-packaged with the robot, much of the information about how to edit the parameters and such were not well described in the online manuals. Although this made for an easier experience for a beginner, a poor understanding of these implementations led to a difficulty troubleshooting issues with the robot without external help.

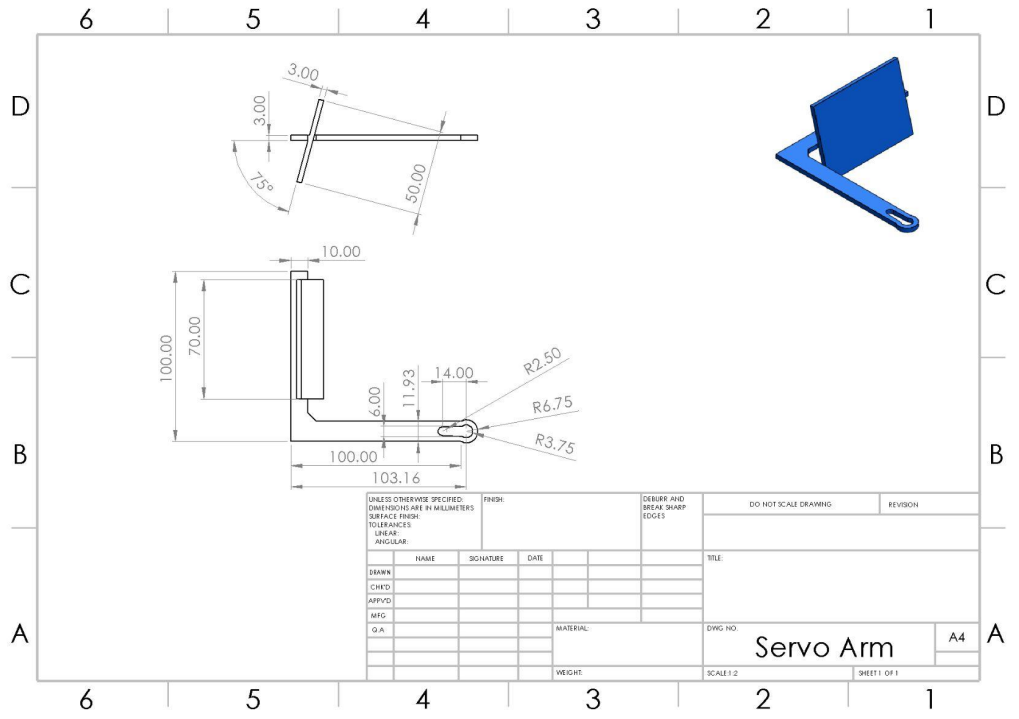
This project was greatly educational and helped to get familiar with and understand the functionality of multi-computer systems and their applications in Mobile Robotics. It also allowed for more knowledge of common problems that need to be solved when dealing with robotics and multi-computer systems in general.

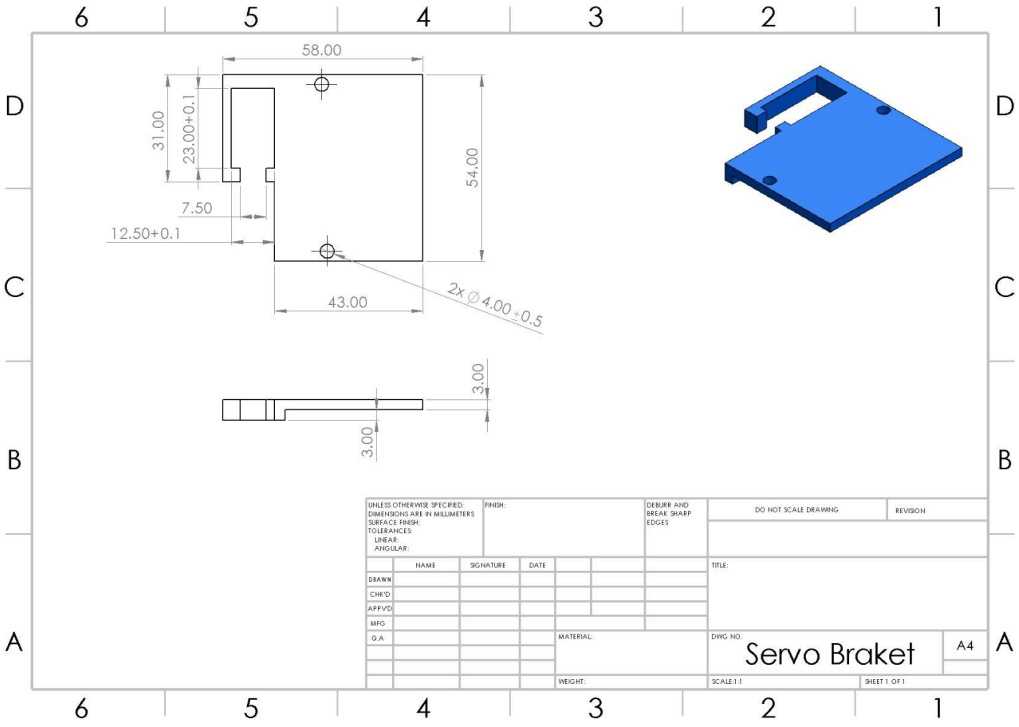
## Appendix

### Part Drawings






## Modified `explore_lite` Source Code

```
#include <explore/explore.h>
#include <thread>
```

```
#include "nav_msgs/Odometry.h"

move_base_msgs::MoveBaseGoal startPositionGoal;
ros::Subscriber posSub;
// gotStart variable to store if a start position is saved yet
bool gotStart = false;

void setPosition(const nav_msgs::Odometry::ConstPtr& msg)
{
  // Do not execute the function if a starting position has already been saved
  if (gotStart) {
    return;
  }
  gotStart = true;
  ROS_INFO("Attempting to save position");
  // Set the start goal using the odometry data from the subscribed topic
  startPositionGoal.target_pose.header.frame_id = "map";
  startPositionGoal.target_pose.header.stamp = ros::Time::now();
  startPositionGoal.target_pose.pose.position.x = msg->pose.pose.position.x;
  startPositionGoal.target_pose.pose.position.y = msg->pose.pose.position.y;
  startPositionGoal.target_pose.pose.orientation.w = 1.0;
}

inline static bool operator==(const geometry_msgs::Point& one,
                              const geometry_msgs::Point& two)
{
  double dx = one.x - two.x;
  double dy = one.y - two.y;
  double dist = sqrt(dx * dx + dy * dy);
  return dist < 0.01;
}

namespace explore {
Explore::Explore()
  : private_nh_("~")
  , tf_listener_(ros::Duration(10.0))
  , costmap_client_(private_nh_, relative_nh_, &tf_listener_)
  , move_base_client_("move_base")
  , prev_distance_(0)
  , last_markers_count_(0) {
  double timeout;
  double min_frontier_size;
  private_nh_.param("planner_frequency", planner_frequency_, 1.0);
  private_nh_.param("progress_timeout", timeout, 30.0);
  progress_timeout_ = ros::Duration(timeout);
  private_nh_.param("visualize", visualize_, false);
  private_nh_.param("potential_scale", potential_scale_, 1e-3);
  private_nh_.param("orientation_scale", orientation_scale_, 0.0);
  private_nh_.param("gain_scale", gain_scale_, 1.0);
  private_nh_.param("min_frontier_size", min_frontier_size, 0.5);

  search_ = frontier_exploration::FrontierSearch(costmap_client_.getCostmap(),
                                                 potential_scale_, gain_scale_,
                                                 min_frontier_size);

  if (visualize_) {
    marker_array_publisher_ =
        private_nh_.advertise<visualization_msgs::MarkerArray>("frontiers", 10);
```

```
  }

  ROS_INFO("Waiting to connect to move_base server");
  move_base_client_.waitForServer();
  ROS_INFO("Connected to move_base server");
  // subscriber to save start position using odom topic
  posSub = relative_nh_.subscribe("odom", 1000, setPosition);
  ROS_INFO("Subscribing to Odom data");

  exploring_timer_ =
      relative_nh_.createTimer(ros::Duration(1. / planner_frequency_),
                               [this](const ros::TimerEvent&) { makePlan(); });
}

Explore::~Explore() {
  stop();
}

void Explore::visualizeFrontiers(
    const std::vector<frontier_exploration::Frontier>& frontiers) {
  std_msgs::ColorRGBA blue;
  blue.r = 0;
  blue.g = 0;
  blue.b = 1.0;
  blue.a = 1.0;
  std_msgs::ColorRGBA red;
  red.r = 1.0;
  red.g = 0;
  red.b = 0;
  red.a = 1.0;
  std_msgs::ColorRGBA green;
  green.r = 0;
  green.g = 1.0;
  green.b = 0;
  green.a = 1.0;

  ROS_DEBUG("visualising %lu frontiers", frontiers.size());
  visualization_msgs::MarkerArray markers_msg;
  std::vector<visualization_msgs::Marker>& markers = markers_msg.markers;
  visualization_msgs::Marker m;

  m.header.frame_id = costmap_client_.getGlobalFrameID();
  m.header.stamp = ros::Time::now();
  m.ns = "frontiers";
  m.scale.x = 1.0;
  m.scale.y = 1.0;
  m.scale.z = 1.0;
  m.color.r = 0;
  m.color.g = 0;
  m.color.b = 255;
  m.color.a = 255;
  // lives forever
  m.lifetime = ros::Duration(0);
  m.frame_locked = true;

  // weighted frontiers are always sorted
  double min_cost = frontiers.empty() ? 0. : frontiers.front().cost;
```

```
  m.action = visualization_msgs::Marker::ADD;
  size_t id = 0;
  for (auto& frontier : frontiers) {
    m.type = visualization_msgs::Marker::POINTS;
    m.id = int(id);
    m.pose.position = {};
    m.scale.x = 0.1;
    m.scale.y = 0.1;
    m.scale.z = 0.1;
    m.points = frontier.points;
    if (goalOnBlacklist(frontier.centroid)) {
      m.color = red;
    } else {
      m.color = blue;
    }
    markers.push_back(m);
    ++id;
    m.type = visualization_msgs::Marker::SPHERE;
    m.id = int(id);
    m.pose.position = frontier.initial;
    // scale frontier according to its cost (costier frontiers will be smaller)
    double scale = std::min(std::abs(min_cost * 0.4 / frontier.cost), 0.5);
    m.scale.x = scale;
    m.scale.y = scale;
    m.scale.z = scale;
    m.points = {};
    m.color = green;
    markers.push_back(m);
    ++id;
  }
  size_t current_markers_count = markers.size();

  // delete previous markers, which are now unused
  m.action = visualization_msgs::Marker::DELETE;
  for (; id < last_markers_count_; ++id) {
    m.id = int(id);
    markers.push_back(m);
  }

  last_markers_count_ = current_markers_count;
  marker_array_publisher_.publish(markers_msg);
}

void Explore::makePlan() {
  // find frontiers
  auto pose = costmap_client_.getRobotPose();
  // get frontiers sorted according to cost
  auto frontiers = search_.searchFrom(pose.position);
  ROS_DEBUG("found %lu frontiers", frontiers.size());
  for (size_t i = 0; i < frontiers.size(); ++i) {
    ROS_DEBUG("frontier %zd cost: %f", i, frontiers[i].cost);
  }

  if (frontiers.empty()) {
    stop();
    return;
  }
```

```
  // publish frontiers as visualization markers
  if (visualize_) {
    visualizeFrontiers(frontiers);
  }

  // find non blacklisted frontier
  auto frontier =
      std::find_if_not(frontiers.begin(), frontiers.end(),
                       [this](const frontier_exploration::Frontier& f) {
                         return goalOnBlacklist(f.centroid);
                       });
  if (frontier == frontiers.end()) {
    stop();
    return;
  }
  geometry_msgs::Point target_position = frontier->centroid;

  // time out if we are not making any progress
  bool same_goal = prev_goal_ == target_position;
  prev_goal_ = target_position;
  if (!same_goal || prev_distance_ > frontier->min_distance) {
    // we have different goal or we made some progress
    last_progress_ = ros::Time::now();
    prev_distance_ = frontier->min_distance;
  }
  // black list if we've made no progress for a long time
  if (ros::Time::now() - last_progress_ > progress_timeout_) {
    frontier_blacklist_.push_back(target_position);
    ROS_DEBUG("Adding current goal to black list");
    makePlan();
    return;
  }

  // we don't need to do anything if we still pursuing the same goal
  if (same_goal) return;

  // send goal to move_base if we have something new to pursue
  move_base_msgs::MoveBaseGoal goal;
  goal.target_pose.pose.position = target_position;
  goal.target_pose.pose.orientation.w = 1.;
  goal.target_pose.header.frame_id = costmap_client_.getGlobalFrameID();
  goal.target_pose.header.stamp = ros::Time::now();
  move_base_client_.sendGoal(
      goal, [this, target_position](
                const actionlib::SimpleClientGoalState& status,
                const move_base_msgs::MoveBaseResultConstPtr& result) {
        reachedGoal(status, result, target_position);
      });
}

bool Explore::goalOnBlacklist(const geometry_msgs::Point& goal)
{
  constexpr static size_t tolerace = 5;
  costmap_2d::Costmap2D* costmap2d = costmap_client_.getCostmap();

  // check if a goal is on the blacklist for goals that we're pursuing
  for (auto& frontier_goal : frontier_blacklist_) {
    double x_diff = fabs(goal.x - frontier_goal.x);
```

```
    double y_diff = fabs(goal.y - frontier_goal.y);

    if (x_diff < tolerace * costmap2d->getResolution() &&
        y_diff < tolerace * costmap2d->getResolution())
      return true;
  }
  return false;
}

void Explore::reachedGoal(const actionlib::SimpleClientGoalState& status,
                          const move_base_msgs::MoveBaseResultConstPtr&,
                          const geometry_msgs::Point& frontier_goal) {
  ROS_DEBUG("Reached goal with status: %s", status.toString().c_str());
  if (status == actionlib::SimpleClientGoalState::ABORTED) {
    frontier_blacklist_.push_back(frontier_goal);
    ROS_DEBUG("Adding current goal to black list");
  }

  // find new goal immediatelly regardless of planning frequency.
  // execute via timer to prevent dead lock in move_base_client (this is
  // callback for sendGoal, which is called in makePlan). the timer must live
  // until callback is executed.
  oneshot_ = relative_nh_.createTimer(
      ros::Duration(0, 0), [this](const ros::TimerEvent&) { makePlan(); },
      true);
}

void Explore::start() {
  exploring_timer_.start();
}

void Explore::stop() {
  move_base_client_.cancelAllGoals();
  exploring_timer_.stop();
  ROS_INFO("Exploration stopped.");
  // after exploration stops, return to start position
  move_base_client_.sendGoal(startPositionGoal);
  move_base_client_.waitForResult();
}

}  // namespace explore

int main(int argc, char** argv) {
  ros::init(argc, argv, "explore");
  if (ros::console::set_logger_level(ROSCONSOLE_DEFAULT_NAME,
                                     ros::console::levels::Debug)) {
    ros::console::notifyLoggerLevelsChanged();
  }
  explore::Explore explore;
  ros::spin();

  return 0;
}
```

**Launch File for Program Execution**

```
<?xml version="1.0"?>
<launch>
    <include file="$(find turtlebot3_slam)/launch/turtlebot3_slam.launch" />
```

```
    <include file="$(find turtlebot3_navigation)/launch/move_base.launch" />
    <include file="/home/bryce/catkin_ws/src/m-explore/explore/launch/explore.launch" />
</launch>
```

## Python Search Node

```
#!/usr/bin/env python

import rospy
import actionlib
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from std_msgs.msg import String
from nav_msgs.msg import OccupancyGrid, Odometry


map_data = None
move_base_client = None
home_goal = None


def xy_to_m2(x, y):
''' Convert an x-y coordinate frame w.r.t. the Occupancy Grid to the map space '''
    width    = map_data.info.width
    height   = map_data.info.height
    x_origin = width  / 2 + map_data.info.origin.position.x
    y_origin = height / 2 + map_data.info.origin.position.y
    new_x    = map_data.info.resolution * (x - x_origin)
    new_y    = map_data.info.resolution * (y - y_origin)
    return (new_x, new_y)


def save_home_odom(msg):
''' Callback to save odom location for return home command '''
    global home_goal
    if home_goal is not None:
        return
    home_goal = MoveBaseGoal()
    home_goal.target_pose.header.frame_id    = "map"
    home_goal.target_pose.pose.position.x    = msg.pose.pose.position.x
    home_goal.target_pose.pose.position.y    = msg.pose.pose.position.y
    home_goal.target_pose.pose.orientation.w = msg.pose.pose.orientation.w


def map_data_callback(data):
    global map_data
    map_data = data


def get_index(x, y, scale):
''' Get the index in the array for a specified row and column '''
    return x + y * scale


def validate_location(x, y, scale, dist_from_walls):
''' Validate a specific location is traversable based on collision detection for nearby obstacles '''
    valid = True
    for x_index in range(x - dist_from_walls, x + dist_from_walls):
```

```
        for y_index in range(y - dist_from_walls, y + dist_from_walls):
            p = map_data.data[get_index(x_index, y_index, scale)]
            if p != 0:
                valid = False
                break
        if not valid:
            break
    return valid


def search_map(map):
''' Pick a set of points in the map based on a cell grid and send the robot to each one to search '''
    row_top     = 0
    row_bottom  = 0
    col_left    = 0
    col_right   = 0
    sector_size = 0.8

    rospy.loginfo("Detecting bounds of search...")
    found_wall  = False
    for i in range(0, map.info.height):
        for j in range(0, map.info.width):
            if (map.data[get_index(j, i, map.info.width)] != -1):
                row_top = i
                found_wall = True
                break
        if found_wall:
            break

    found_wall = False
    for i in range(map.info.height - 1, 0, -1):
        for j in range(map.info.width - 1, 0, -1):
            if map.data[get_index(j, i, map.info.width)] != -1:
                row_bottom = i
                found_wall = True
                break
        if found_wall:
            break

    found_wall = False
    for j in range(map.info.width - 1, 0, -1):
        for i in range(map.info.height - 1, 0, -1):
            if map.data[get_index(j, i, map.info.width)] != -1:
                col_right = j
                found_wall = True
                break
        if found_wall:
            break

    found_wall = False
    for j in range(0, map.info.width):
        for i in range(0, map.info.height):
            if map.data[get_index(j, i, map.info.width) != -1]:
                col_left = j
                found_wall = True
                break
        if found_wall:
            break
```

```
    rospy.loginfo("Search bounds located. Beginning sector sweep.")
    for col in range(col_left, col_right, sector_size / map.info.resolution):
        for row in range(row_top, row_bottom, sector_size / map.info.resolution):
            if validate_location(round(col), round(row), map.info.width, 3):
                x, y = xy_to_m2(col, row)
                rospy("Searching location (" + str(x) + ", " + str(y) + ")...")
                goal = MoveBaseGoal()
                goal.target_pose.header.frame_id    = "map"
                goal.target_pose.header.stamp       = rospy.Time.now()
                goal.target_pose.pose.position.x    = x
                goal.target_pose.pose.position.y    = y
                goal.target_pose.pose.orientation.w = 1.0
                move_base_client.send_goal(goal)
                move_base_client.wait_for_result()

    rospy.loginfo("Search complete. Returning home.")
    home_goal.target_pose.header.stamp = rospy.Time.now()
    move_base_client.send_goal(home_goal)
    move_base_client.wait_for_result()


def exploration_finished_callback(msg):
    global map_data
    map_data = msg
    rospy.loginfo("Exploration complete. Beginning search.")
    search_map(map_data)


def main_node():
    map_sub = rospy.Subscriber('map', OccupancyGrid, map_data_callback)
    exp_sub = rospy.Subscriber('exploration_msgs', String, exploration_finished_callback)
    hom_sub = rospy.Subscriber('odom', Odometry, save_home_odom)
    rospy.init_node('fiducial_search_node', anonymous=True)
    global move_base_client
    move_base_client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
    rospy.loginfo("Search node started. Waiting for move_base client...")
    move_base_client.wait_for_server()
    rospy.loginfo("Client detected (move_base). Waiting for exploration to complete...")
    rospy.spin()


if __name__ == '__main__':
    try:
        main_node()
    except rospy.ROSInterruptException:
        pass
```

## RViz Marker Publisher Node

```
#include <ros/ros.h>
#include <tf2_ros/transform_listener.h>
#include <geometry_msgs/TransformStamped.h>
#include <visualization_msgs/Marker.h>
```

```cpp
#include <cstdlib>

int main(int argc, char** argv) {
  ros::init(argc, argv, "fiducial_publisher_node");
  ros::NodeHandle _nh("~");
  std::string param;
  int code_num;
  _nh.getParam("fiducial_code", param);
  code_num = atoi(param.substr(9, 12).c_str());
  ROS_INFO(param.c_str());

  ros::Publisher vis_pub = _nh.advertise<visualization_msgs::Marker>("visualization_marker", 0);
  visualization_msgs::Marker marker;
  marker.header.frame_id = "map";
  marker.id = code_num;
  marker.type = visualization_msgs::Marker::SPHERE;
  marker.action = visualization_msgs::Marker::ADD;
  marker.scale.x = 0.1;
  marker.scale.y = 0.1;
  marker.scale.z = 0.1;
  marker.pose.orientation.x = 0.0;
  marker.pose.orientation.y = 0.0;
  marker.pose.orientation.z = 0.0;
  marker.pose.orientation.w = 1.0;
  marker.color.a = 1.0;
  marker.color.r = 0.5 * ((double) code_num - 100.0);
  marker.color.g = 1.0 - 0.5 * ((double) code_num - 100.0);
  marker.color.b = 0.2;

  tf2_ros::Buffer tfBuffer;
  tf2_ros::TransformListener tfListener(tfBuffer);

  ros::Rate rate(10.0);
  while (_nh.ok()) {
    geometry_msgs::TransformStamped transform;
    try {
      transform = tfBuffer.lookupTransform("map", param.c_str(), ros::Time(0));
    } catch (tf2::TransformException &ex) {
      ros::Duration(0.5).sleep();
      continue;
    }
    marker.pose.position.x = transform.transform.translation.x;
    marker.pose.position.y = transform.transform.translation.y;
    marker.pose.position.z = transform.transform.translation.z;
    vis_pub.publish(marker);
    rate.sleep();
  }

  return 0;
}
```

**Point Cloud Publisher Node**

```python
#!/usr/bin/env python
import rospy
import math
import tf2_ros
```

```
import logging

from sensor_msgs.msg import PointCloud2
import std_msgs.msg
import sensor_msgs.point_cloud2 as pcl2


inflation = 0.3
circle_points = 25.0
pkg1 = None
pkg2 = None


def make_circle(x, y):
    cloud_points = []
    for i in range(0, int(circle_points)):
        cloud_points.append(
            [
                x + inflation / 2 * math.sin(2 * float(i) / circle_points * math.pi),
                y + inflation / 2 * math.cos(2 * float(i) / circle_points * math.pi),
                0.1
            ]
        )
    return cloud_points


def main():
    global pkg1
    global pkg2
    rospy.init_node('avoid_packages_node')
    pcl_pub = rospy.Publisher("/package_cloud", PointCloud2, queue_size=100)
    rospy.loginfo("Initializing package avoidance node...")
    rospy.sleep(1.0)

    tf_buffer = tf2_ros.Buffer()
    tf_listener = tf2_ros.TransformListener(tf_buffer)

    header = std_msgs.msg.Header()
    header.stamp = rospy.Time.now()
    header.frame_id = 'map'

    rate = rospy.Rate(1)
    while not rospy.is_shutdown():
        cloud_points = []
        try:
            rospy.sleep(0.01)
            pkg1 = tf_buffer.lookup_transform("map", "fiducial_101", rospy.Time())
            rospy.loginfo("Transform 101 successfully located")
            cloud_points.append(make_circle(pkg1.transform.translation.x,
pkg1.transform.translation.y))
        except Exception as Argument:
            if pkg1 is not None:
                cloud_points.append(make_circle(pkg1.transform.translation.x,
pkg1.transform.translation.y))
            else:
                rospy.loginfo("Fiducial 101 is not yet detected. " + str(Argument))
        finally:
            try:
```

```
                rospy.sleep(0.01)
                pkg2 = tf_buffer.lookup_transform("map", "fiducial_100", rospy.Time())
                rospy.loginfo("Transform 100 successfully located")
                cloud_points.append(make_circle(pkg2.transform.translation.x,
pkg2.transform.translation.y))
            except:
                if pkg2 is not None:
                    cloud_points.append(make_circle(pkg2.transform.translation.x,
pkg2.transform.translation.y))
                rospy.loginfo("Fiducial 100 is not yet detected. " + str(Argument))
        rospy.loginfo(cloud_points)
        rospy.loginfo(pkg1)
        pcl_pkg = None
        if pkg1 and pkg2:
            pcl_pkg = pcl2.create_cloud_xyz32(
                header,
                make_circle(
                    pkg1.transform.translation.x,
                    pkg1.transform.translation.y
                ) +
                make_circle(
                    pkg2.transform.translation.x,
                    pkg2.transform.translation.y
                )
            )
        elif pkg1:
            pcl_pkg = pcl2.create_cloud_xyz32(
                header,
                make_circle(
                    pkg1.transform.translation.x,
                    pkg1.transform.translation.y
                )
            )
        elif pkg2:
            pcl_pkg = pcl2.create_cloud_xyz32(
                header,
                make_circle(
                    pkg2.transform.translation.x,
                    pkg2.transform.translation.y
                )
            )
        if pcl_pkg:
            pcl_pub.publish(pcl_pkg)
        rate.sleep()


if __name__ == '__main__':
    try:
        main()
    except rospy.exceptions.ROSInterruptException:
        pass
```

**References**


[1] Yuan Xu, Y. S. Shmaliy, Yueyang Li, Xiyuan Chen, and Hang Guo, "Indoor INS/LiDAR-Based Robot Localization With Improved Robustness Using Cascaded FIR Filter," *IEEE Access*, vol. 7, pp. 34189–34197, Mar. 2019.

[2] Kai-Tai Song *et al.*, "Navigation Control Design of a Mobile Robot by Integrating Obstacle Avoidance and LiDAR SLAM," *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 1833–1838, Oct. 2018.

[3] J. Hörner, "Map-merging for multi-robot system" B.S. thesis, Dept. of Theo. Comp. Sci. Math. Logic, Charles Univ., Prague, Czechia, 2016. [Online]. Available: https://dspace.cuni.cz/handle/20.500.11956/83769

[4] H. Umari and S. Mukhopadhyay, "Autonomous robotic exploration based on multiple rapidly-exploring randomized trees," *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1396–1402, Sep. 2017.

[5] F. Novotny, "Wiki," *ros.org*, 16-Aug-2016. [Online]. Available: http://wiki.ros.org/visp_auto_tracker. [Accessed: 22-Mar-2022].

[6] J. Vaughan, "Wiki," *ros.org*, 13-Jul-2020. [Online]. Available: http://wiki.ros.org/aruco_detect. [Accessed: 22-Mar-2022].