EECE 5550: Mobile Robotics Final Project OOGWAY, an Autonomous Disaster Response Reconnaissance Robot

Abdulrahman Al-Badawi
MS Robotics
Northeastern University
Boston, USA
al-badawi.a@northeastern.edu

Hariram Arni
MS Robotics
Northeastern University
Boston, USA
arni.h@northeastern.edu

Prem Sukhadwala
MS Robotics
Northeastern University
Boston, USA
sukhadwala.p@northeastern.edu

Guangping Liu
MS Mechanical Engineering
Northeastern University
Boston, USA
liu.guangp@northeastern.edu

Joy Fakhry
MS Robotics
Northeastern University
Boston, USA
fakhry.jo@northeastern.edu

Sriram Kodeeswaran
MS Robotics
Northeastern University
Boston, USA
kodeeswaran.s@northeastern.edu

***Abstract*—The goal of this project was to develop an autonomous reconnaissance robot, using TurtleBot3, to map and search for victims–simulated using AprilTags–in disaster-stricken areas, providing crucial information to human operators. We used GMapping for SLAM, OpenCV for people detection, and the explore_lite and apriltag ROS packages for exploration and AprilTag detection, respectively. After a month of development, the robot successfully provided an accurate occupancy grid map of the environment. It also detected all of the AprilTags, however we were unable to produce accurate pose estimations.**

***Index Terms*—turtlebot, ROS, AprilTag, autonomous exploration, path planning**

I. Motivation and Introduction

Robots are expensive, but not as expensive as a priceless human life. If a robot can achieve the same goal as a human, it frees them up to do something else that is more useful, or not as dangerous. As technology has improved, humans have sent robots into increasingly hazardous environments to keep search and rescue personnel safe and aid in finding victims. We have seen exponential automation growth in all industries over the last century, and disaster response and reconnaissance is the latest field to get included in that growth. According to a survey shown in [1], ground robots make up the majority of robots deployed at disaster sites. A popular example is Inuktun's VGTV Xtreme, used after hurricane Katrina [2]. More recently, robots like GuardianS [3] and Vine robots [4] from Stanford University are at the forefront of helping humans with search and rescue operations.

To learn more about mobile robotics with hands-on experience and implement some of the class theory on a real robot, our team elected to create our own little autonomous disaster response reconnaissance robot, dubbed Oogway, after the great and wise protector of the Valley of Peace from Kung Fu Panda, who also happens to be a tortoise. Oogway implements laser and odometry based Simultaneous Localization And Mapping (SLAM) to get an accurate belief of its position and surroundings, cost- based path planning and computer vision techniques to detect AprilTags and/or people.

Oogway was built on the TurtleBot3 Burger platform, which comes with a 2D Lidar scanner, high-definition RaspberryPi camera, two Dynamixel motors, an OpenCR board to assist with running ROS, and a Raspberry Pi for our Ubuntu OS. The goal of Oogway is to remotely and autonomously map a complicated environment and detect victims, simulated by AprilTags, reporting back the final occupancy grid, as well as the tag IDs and locations.

II. Proposed Solution

Our solution is described in the subsequent three subsections, based on the main areas our team implemented. Aspects like SLAM, visual object detection, control, and path planning were all handled by pre-built and optimized packages. Our implementation mostly consisted of connecting these systems onto a single, relatively easy to use platform.

A. Victim Detection

We have attempted two forms of victim detection for this project. One, where simulated victims are represented as AprilTags, and another where we use object detection to recognize real humans.

1) AprilTag Detection and Localization: The AprilTag portion of the system has two main sub-problems. First, we need to detect the tag, and second, we need to convert the pose of that detection into the map frame. The first part was handled by the apriltag_ros package. The detector uses as inputs, the raw images from the Raspberry Pi camera as well as its intrinsic

parameters (obtained using the camera_calibration package), and outputs the tag ID's and pose estimates with respect to the camera frame. The package has various parameters that affect tag detection accuracy and speed, such as image decimate to speed up detection by cropping the image, or refine edges to improve pose estimation accuracy. The effect of these parameters are discussed in the results section.

The second part of the AprilTag subsystem was handled primarily through the tf package. The tags' pose estimates were published as a topic which we converted to frames. This introduced each tag's pose into the tf ecosystem. From there, we formed a link between the tag pose relative to the camera, to the map. In this link, we also rotated the axes from the z-forward camera coordinate system to be z-up to coincide with the map coordinate system. Using the broadcasted link from tag to map, we could simply subscribe to the /tf topic to obtain the full transform.

For reference, the AprilTag sub-system is outlined below in Fig. 1.

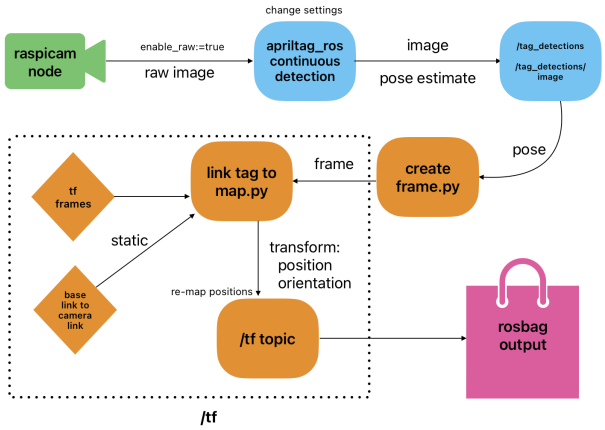

Fig. 1. AprilTag System Block Diagram

2) Human Detection: AprilTags are simulated victims. That makes the system easier to implement and test. Since we achieved AprilTag detection successfully, we aimed to expand the robot's capabilities by integrating people detection into its vision system. To achieve this goal, we utilized the powerful computer vision library OpenCV and implemented a Python script that enables the Turtlebot to detect people in its surroundings. With this new functionality, the robot can now not only identify and track objects with AprilTags but also recognize humans in its environment.

When the application is started, Raspicam subscribes to the ROS topic usb_cam/image_raw, which is where the turtlebot's camera transmits raw image data. When the people_detection package receives the image data, it processes it using the OpenCV library to detect any humans in the image. The application specifically uses the HOG (Histogram of Oriented Gradients) descriptor technique to detect sections of the image that include individuals. When a person is spotted, the computer draws a bounding box around them to highlight their location on the image. The processed image is then converted back into a ROS-compatible format and published to the /processed_image topic.

By executing this program continually, the turtlebot may detect persons in its range of view and alert the user by publishing the processed image to the /processed_image topic. The user can then examine the processed image to view the detected people and their locations.

This development increases the real world capability of our system by integrating a more realistic victim detection system.

B. Navigation

For this project, we wanted Oogway to be able to navigate autonomously in its environment and also create maps of its surroundings. We used GMapping, a package in ROS, to create a map using SLAM. GMapping takes in data from the robot's laser scans and odometry, and outputs a 2D occupancy map.

We used the move_base ROS package to move around the robot, which is a part of our ROS navigation stack. This package takes in information from GMapping and uses it to help Oogway navigate to specific locations in the map. The move_base package uses global and local navigation planners to do path planning and obstacle avoidance. It calculates the shortest path for Oogway to take based on the information it has about the environment, such as the location of obstacles and the goal.

By combining the explore_lite and move_base ROS packages, we were able to develop an autonomous mapping and navigation system for Oogway. The system allowed Oogway to create accurate maps of its environment and move around autonomously, even in the presence of obstacles. In Fig. 2, the white area represents known-free space, the grey and black areas represent unknown and known-occupied space respectively, while the blue square shows the local-costmap and the blue blob indicates the global-costmap. The local planner creates a local path using the local-costmap to avoid obstacles, shown in yellow. The global planner creates a global path based on the global-costmap, shown in red and black. The lidar points are shown in green.

C. Exploration

To explore an unknown and hazardous environment, we used the explore_lite ROS package that provides greedy frontier-based exploration. Frontiers can be described as boundaries between known and unknown cells on a map. To understand this package, the system can be separated into two sections–frontier detection and path planning.

For frontier detection, explore_lite takes in the map of the environment that keeps updating with each timestamp. It enumerates all the frontiers on the map using a Breadth First Search algorithm, calculates each frontier's centroid, and assigns costs based on the distance from the robot and the cost map given to it by move_base.

While doing the frontier search, the package simultaneously sends the coordinates of frontiers' centroids as goals to the move_base node we used for navigation. This will move the robot toward the frontiers and explore a map area that was

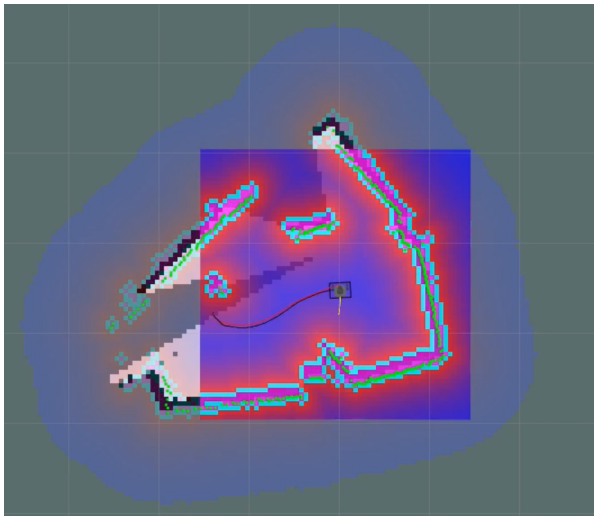

Fig. 2. Mapping and Navigation in RViz

previously unknown. It will keep doing this until we have a complete map with no frontiers left. Path planning gets taken care of by the move_base node using the global and local cost maps.

With this, Oogway can autonomously explore and navigate in an unknown environment. However, we need to consider whether it will detect all the AprilTags or victims in an environment. The camera used for detection is in front of the robot. It has a limited field of view, while the lidar has a range of 360. The robot can map faster and at longer distances but the camera would not be able to look around as much. It will fail to detect a lot of victims. Therefore, a basic implementation of navigation and exploration packages will not guarantee to search for all the victims or AprilTags. To address this issue, we devised the subsequent three approaches.

1) Rotate Interrupt Service: We developed a ROS service node that interrupts the move_base node after every 15 seconds to rotate the robot for a full circle before resuming its original path. The idea is that this will allow the camera to be able to look around more in the environment, hence having a higher chance of finding more victims or AprilTags.

The *Turn360Degrees* function in Algorithm 1 is our ROS service node. If a current goal exists, the node interrupts the exploration and stops the robot from moving toward it.

The function then initializes the necessary variables and publishers for controlling the robot's angular velocity. The angular speed is set according to the user's preference, and the duration needed to complete a full rotation is calculated. The loop publishes a Twist message until the robot completes the turn.

Upon completion, the angular speed is set to 0 to stop the rotation, and the Twist message is published again to ensure the robot comes to a halt.

Finally, the previous goal is republished allowing the robot to resume its navigation toward its original destination.

Algorithm 1 Interrupt Service

Input: Empty Request
Output: Empty Response

function *Turn360Degrees*
  if current_goal is None **then**
    return EmptyResponse()
  end if
  Cancel current goal
  Initialize *vel_pub* ← new Twist publisher
  Set angular_speed = 1 rad/s
  Compute duration = $\frac{2\pi}{\text{angular\_speed}}$
  Initialize twist with angular.z ← angular_speed
  Initialize start_time ← current time
  while elapsed time < duration **do**
    Publish twist ← vel_pub
    Sleep for a short duration
  end while
  Set twist.angular.z ← 0
  Publish twist ← vel_pub
  Publish current_goal again.
  return EmptyResponse()
end function

The primary aim for the client interrupt, shown in Algorithm 2 is to act as a client for the interrupt service, periodically calling the service to perform the 360-degree rotation.

Algorithm 2 Interrupt Service Client

function *callservice*
  Wait for interrupt service to be available
  Try to:
    Create a service proxy for "interrupt service"
    Call the service and store the response
    Return "Success" status from response
  If Exception:
    Print "Failed" status from response
  if __name__ == '__main__':
    Initialize interrupt client ROS node.
    Set loop rate of 0.06 Hz (every 15 seconds)
  Enter loop until node is shutdown:
    Call the service
    Print "Success" status of the service call
    Sleep for a duration specified by loop rate
end function

Advantages:

- Slightly improving the chance of finding AprilTags

Disadvantages:

- Robot will finish mapping before it can detect all the AprilTags
- Will not go into a confined corner if it can map it from the outside

2) Lidar Range Manipulation: As discussed earlier, the camera has a limited field of view in front of the robot and the lidar sensor has a range of 360°. So, with this approach, we decided to match the range of two sensors by cutting the angular range of the lidar sensor. For this, we developed a separate node that will subscribe to the lidar topic /scan, and using a Python file, filter and only pass on the data that lies in the front of the robot at a 60° angle. Fig. 3 shows the lidar sensor's angular range with respect to the robot.

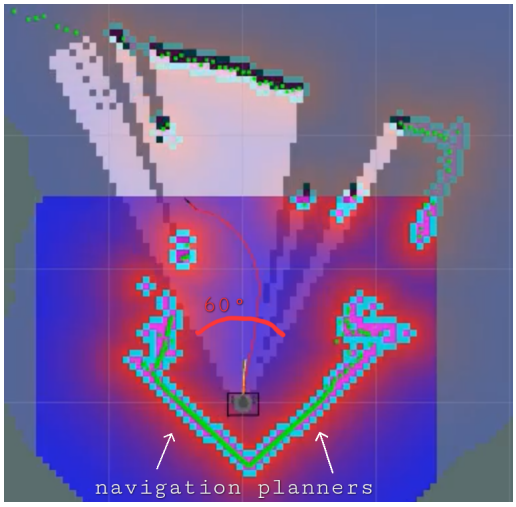

Fig. 3. RViz output displaying Lidar scan using manipulated ranges

To generate a complete map, the robot needs to look around more in the environment, since it doesn't have a 360° lidar view. However, it still needs to see obstacles to move safely. To address this, only the SLAM node receives filtered /scan data while move_base receives the full /scan data. Despite the lidar sensor's limited angular range in front of the robot, move_base's navigation planners can still detect objects behind it, as shown in Fig. 3.

Advantages:

- The robot will find almost all if not all AprilTags.

Disadvantages:

- The robot does not receive enough /scan data to have an accurate belief of the map and/or its location which results in a bad map in the end as shown in Fig 4.

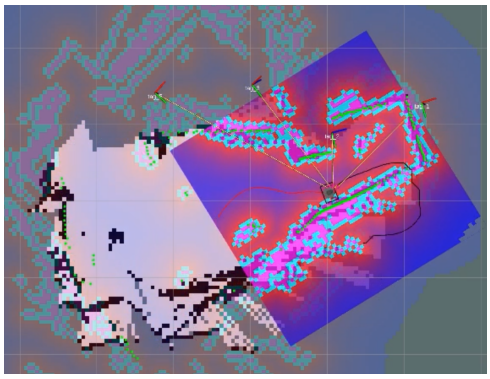

Fig. 4. Inaccurate mapping with filtered Lidar data

3) Randomized Sampling-Based Post-Exploration: Our approach differs from the previous two, which involved the robot searching for AprilTags during mapping. Instead, we allowed the robot to complete the mapping process, detecting any AprilTags in the process, and then conducted a separate search for additional tags. After the complete map is received, we implement a method for sampling various points within the map, which helps identify any tags missed during the initial exploration phase.

By incorporating the map information, the robot is efficiently able to navigate to the sampled points. The robot moves to each randomly sampled point, performs a 360-degree rotation, and proceeds to the next point. This rotation is adapted from the interrupt service algorithm, allowing the robot to thoroughly scan its surroundings at each point. Traversing in the environment multiple times and rotating the camera around at randomly sampled points increases the likelihood of detecting any missed AprilTags.

To maintain efficiency, if the robot is unable to reach a point within a 20-second time limit, it abandons the current goal and moves on to the next point. This approach ensures that the robot does not waste time on unreachable points and can cover more areas within a reasonable timeframe. For reference, the random sampling-based exploration sub-system is illustrated in Fig. 5.

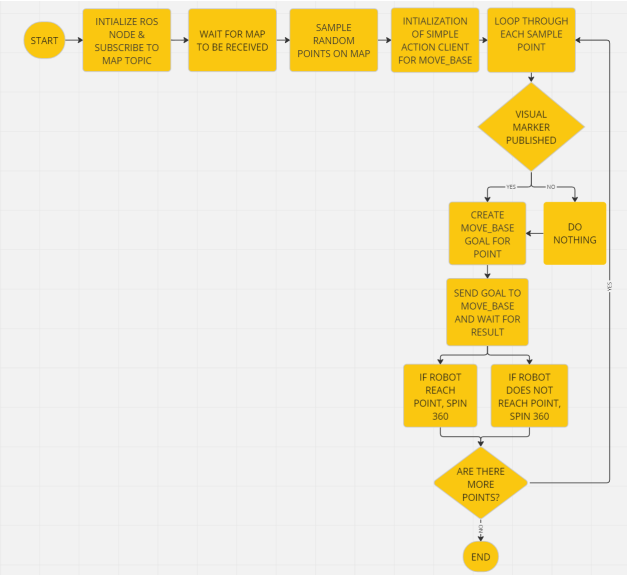

Fig. 5. Random Sampling-Based Post-Exploration Block Diagram

Algorithm 3 iterates until the desired number of points are obtained, generating random coordinates and checking if they represent free spaces on the map explored during the initial phase. If the coordinate is a free space in the map the point is added to the list.

Algorithm 3 Random Sampling

Input: An integer *num_points* representing the number of random points to sample.

Output: A list of *Point* objects representing the sampled random points.

function *SampleRandomPoints*(num_points)
  Initialize set *points* to empty list
  while *len(points)* $<$ *num_points* **do**
    Generate a random integer x from $(0\text{-}mapwidth)$
    Generate a random integer y from $(0\text{-}mapheight)$
    if map_data[y][x] = 0 (indicating free space) **then**
      Compute *px*,*py* $\leftarrow$ map resolution and origin
      Create a *Point* with coordinates (*px*, *py*, 0.0)
      Append to the list *points*.
    end if
  end while
  return the list *points*
end function

Advantages:

- Likelihood of finding all the AprilTags in the environment is very high
- Can simply increase *num_points* to increase coverage of environment
- Large areas with open space are more likely to get sampled than small areas with possibly complex terrain, meaning complicated areas are not going to get view as much

Disadvantages:

- Since the samples are random, we can get multiple points in the same neighborhood of the map
- Can be time consuming which is a crucial factor in search and rescue

D. Simulation

Gazebo simulations provide a controlled and consistent environment, enabling developers to accurately compare the performance of various algorithms, settings, or behaviors during testing. Simulation is important for testing the software components, robot behavior and planning algorithms in different surrounding environments. In this section we introduce a simulation environment for mobile robots based on ROS and Gazebo [5]. The primary goal of this simulation is to thoroughly evaluate the TurtleBot within a controlled setting before transitioning to real-world testing.

The Randomized Sampling-Based Exploration was effectively tested in Gazebo using a world map. The explore_lite package was initially launched to fully map the environment. Once the environment mapping was complete, the exploration algorithm was executed to sample random points throughout the world map.

As demonstrated in Fig. 6, visualization markers were used to indicate the random points, employing the publish_marker() function. This function generates a sphere marker at each point's location and publishes it to the /visualization_marker topic. The robot proceeds to navigate to each sampled point, performing a 360-degree turn to thoroughly scan the surroundings before moving on to the next point.

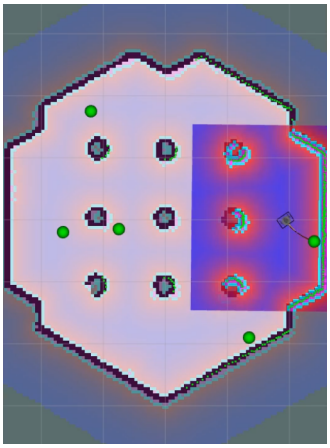

Fig. 6. Simulation in a Controlled Environment

III. Results

By implementing SLAM, we obtained a clear and complete 2D map shown in Fig. 7. This occupancy grid accurately describes the correct outline and arrangement of the test environment, providing a reliable resource for people in unknown area exploration.

Utilizing the system as described above, we were able to successfully navigate and explore our environment, and detect AprilTags. We used the random sampling technique to further explore the environment after it has created an initial map. This further improved the map, and the final version is shown below in Fig. 7.

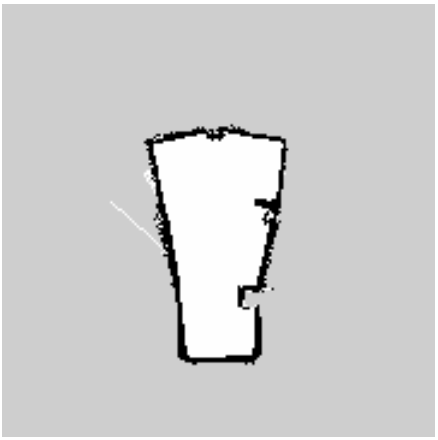

Fig. 7. Final environment occupancy grid

Fig. 8 below is a plot of the average detection location for each AprilTag overlaid on the final map, compared to the ground truth location for each tag. The dashed lines in Fig. 8 serve as a visual indicator for the distance between where our system thinks each tag is to where it actually is. As shown, the results are quite poor.

We tried changing the aforementioned AprilTag detector parameters, which had significant effects when the tag detector was tested on a stationary robot, but as soon as we tried moving the robot, the results came out looking like noise relative to the map frame, regardless of the parameter values. We believe this is primarily due to mapping errors. One source of error arises due to detections occurring while the map is still being constructed. While initially exploring, the robot has not yet created a loop closure to update the map, which means its belief of the map could be quite far off from the ground truth. Since our implementation of pose detection for the tags depends on the current map at the time of detection, if the map is wrong, the tag pose will be wrong as well.

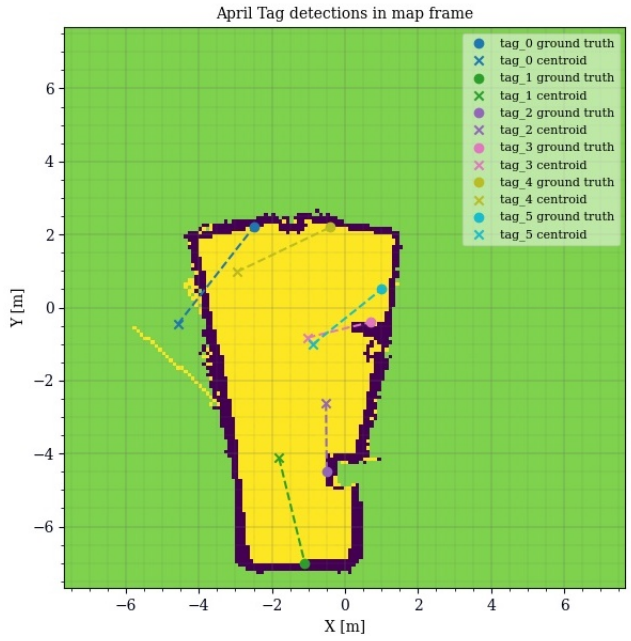

Fig. 8. AprilTag detection estimates and ground truth locations in map frame

IV. Discussion

A. Challenges

- Running cartographer seemed to produce an error with URDF due to missing transforms
- Initially, explore_lite did not work as intended as the turtlebot would simply rotate 360° , in place, approximately every 10 seconds
- No AprilTags were detected after roslaunching the launch file“continuous_detection.launch”, inside the apriltag_ros package
- April Tag detection, during initial testing, was less than ideal
- While attempting a catkin_make on the raspicam package, the process often froze
- Poor internet connection or changing IP addresses, resulted in failure of bringup

B. Learnings

- Understand the workings of a lidar scanner, and gain the ability to manipulate lidar scan data
- Evaluate the turtlebot’s performance in a controlled environment(Gazebo)
- Best practices for working with launch files, creating publishers, subscribers, services, and clients, among other ROS-related tasks
- Prioritize data over hotspot, e.g. adding a router to the bot

V. Conclusion

In summary, our project involved implementing SLAM using GMapping and explore_lite to map and explore the test area. We detected victims using AprilTags through two different methods: lidar field of view manipulation and random sampling. Among these approaches, random sampling proved to be more robust compared to the lidar manipulation method. As an extension of our project, we detected people with OpenCV in ROS..

In order to improve our accuracy in tagging, one of our upcoming plans is to consider loop-closure and changing belief of the map and the robot’s position. The belief over the map and the robot’s position keeps getting accurate with more information and loop-closures. Thus we can update AprilTags’ poses after loop-closure and achieve accurate information about the map. We believe our implementation of random sampling was a good first attempt, however many optimizations and upgrades can be made. For example, we could use importance sampling, or sample locations based on our knowledge of the map, to make the search more efficient.

With these improvements, our project can continue to evolve and provide even better results in detecting victims and navigating unknown environments.

VI. Technical Contribution

- Abdul: Navigation (move_base), explore_lite, gmapping
- Guangping: Navigation (move_base), AprilTag Transformation
- Hariram: Gazebo simulation, rotate service, and random sampling
- Prem: Navigation, explore_lite, rotate service, lidar manipulation, and a small portion of AprilTags
- Joy: AprilTag detection, transformations, and analysis
- Sriram: AprilTag detection, object detection

VII. Project Links

Code: https://gitlab.com/joyfakhry/mobile-robotics-oogway
Video: https://youtu.be/lUepFLHuZ8g

References

[1] B. Siciliano, O Khatib, et. al, “Springer Handbook of Robotics” Springer Cham, 2nd Ed, pp. 1583, 2016.

[2] M. Micire, ”VGTV-Xtreme robot used during search after Hurricane Katrina”, *UMass Lowell Robotics Lab*, Sep 1, 2009, Available: https://www.youtube.com/watch?v=EuTpUa7jgJ8.

[3] Sarcos Technology and Robotics Corporation, ”Guardian S: Urban Search and Rescue”, *Sarcos Technology and Robotics Corporation*, Feb 14, 2019, Available: https://www.youtube.com/watch?v=03j0P0Pt_l0.

[4] A. Ward, ””Growing Robots That Can be Used for Search and Rescue — Innovation Nation””, *The Henry Ford*, May 30, 2020, Available: http://www.youtube.com/watch?v=jlaNjPi4hCo&t=4s.

[5] K. Takaya, T. Asai, V. Kroumov and F. Smarandache, ”Simulation environment for mobile robots testing using ROS and Gazebo,” 2016 20th International Conference on System Theory, Control and Computing (ICSTCC), Sinaia, Romania, 2016, pp. 96-101, doi: 10.1109/ICSTCC.2016.7790647.