Señor Turtle: Search and Rescue

Febin Wilson
wilson.fe@northeastern.edu

Adnan Amir
amir.ad@northeastern.edu

Ramez Mubarak
ramez.s.mubarak@gmail.com

***Abstract*—This project explores the integration of a modified Model Predictive Path Integral (m-MPPI) control algorithm into a robotic search and rescue operation using the Turtlebot3 platform. Utilizing the Turtlebot3 SLAM package, the robot autonomously maps an environment and identifies unexplored frontiers. These frontiers serve as goals for the m-MPPI control algorithm, which calculates optimal paths to navigate while continuing to map the surroundings. The application of m-MPPI, alongside Cartographer SLAM technology, demonstrates significant improvements in the robot's navigation and mapping accuracy, enhancing its capabilities for effective search and rescue operations.**

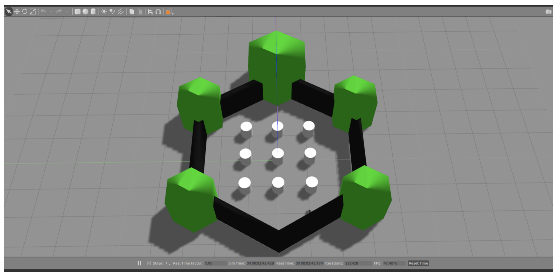


Fig. 1. Turtlebot3 world

I. Introduction

The objective of this project is to enhance robotic capabilities in search and rescue missions through advanced navigation and mapping technologies. The Turtlebot3 platform is equipped with sensors and software that enable real-time localization and mapping of complex environments, a critical feature in rescue scenarios where precision and reliability are paramount. The integration of the m-MPPI control algorithm allows the robot to navigate efficiently by optimizing trajectory paths dynamically, considering both current and predictive state estimations. This project builds on foundational work in simultaneous localization and mapping (SLAM), specifically utilizing the Cartographer SLAM framework developed by Google. This framework supports robust 2D and 3D mapping, crucial for the diverse environments encountered during search and rescue operations. By combining these technologies, this project aims to develop a robotic system capable of performing autonomous search and rescue tasks with increased efficiency and decreased human intervention.

II. SLAM System

In this project, Cartographer SLAM technology plays a crucial role by enabling real-time simultaneous localization and mapping (SLAM) for the Turtlebot3 robotic platform. This system empowers the robot to navigate complex environments efficiently, enhancing its capability in search and rescue operations.

A. Relevance to Search and Rescue

The ability of Cartographer SLAM to create detailed 2D and 3D maps is essential for search and rescue missions, where rapid and accurate environmental assessment is crucial. The technology ensures that the robotic platform can navigate debris-filled or unfamiliar terrain by providing precise and up-to-date mapping information, facilitating effective search strategies and obstacle avoidance. The fig 1 shows the simulation environments for turtlebot3. In search and rescue missions, environments are often chaotic and unpredictable due to natural disasters or accidents. Cartographer SLAM enables robots like Señor Turtle to dynamically adjust their navigation strategies by creating accurate, real-time maps.

B. Functional Mechanism

Cartographer SLAM combines data from various sensors, including LiDARs and IMUs, to produce accurate maps that are critical for navigating and strategizing in dynamic or unpredictable environments typical of search and rescue scenarios. Its loop closure detection feature is particularly valuable, allowing the system to correct any drift in the map by recognizing previously visited locations, thereby maintaining the integrity and accuracy of the map throughout the mission.

Cartographer SLAM harnesses data from multiple sensors equipped on the Turtlebot3, primarily LiDAR and IMU (Inertial Measurement Unit). The LiDAR provides detailed 360-degree scans to detect obstacles and map the environment, while the IMU tracks the robot's orientation and acceleration, crucial for maintaining the correct posture and navigation accuracy under varying terrain conditions.

The front end of the system, or Local SLAM, processes incoming sensor data to create local submaps. These submaps consist of small, manageable areas that are quickly processed to reflect the immediate surroundings of the robot. This step is essential for maintaining real-time responsiveness in dynamic environments where situational awareness is constantly changing. Each submap is a locally consistent representation of the environment. As Señor Turtle moves through a search area, new submaps are continuously generated, overlapping with previous ones to ensure comprehensive coverage and map integrity. The fig 2 shows an inflation radius. This parameter makes inflation area from the obstacles.

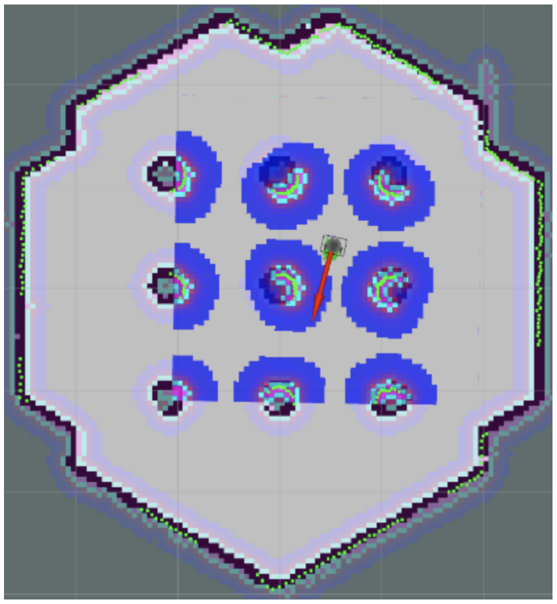


Fig. 2. Inflation radius

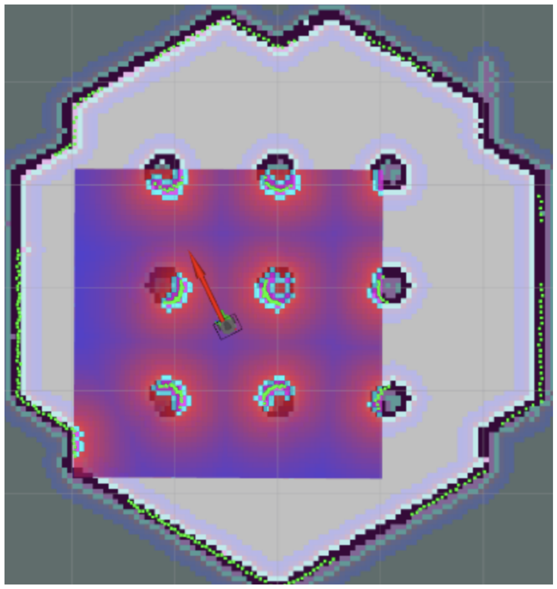


Fig. 3. Cost Scaling Factor

An integral component of the backend, Global SLAM, involves identifying places the robot has previously visited (loop closure). By recognizing these areas, Cartographer SLAM can adjust for cumulative mapping errors and drift, refining the overall map accuracy. This process integrates all individual submaps into a cohesive global map. The backend uses sophisticated algorithms to stitch these submaps together, ensuring that they align correctly based on the detected loop closures and the overall geometry of the mapped area.

Utilizing the map generated by Cartographer SLAM, the path planning system within the Turtlebot3 calculates optimal routes to unexplored areas or to specific coordinates where search efforts are concentrated. The paths are dynamically updated in response to new obstacles detected by the ongoing SLAM process. With real-time updates, the robot can react promptly to avoid sudden obstacles, such as falling debris or unexpected terrain changes, which are common in rescue scenarios.

The maps and navigation paths generated by Cartographer SLAM can be shared in real-time with rescue coordinators and other automated systems, facilitating a coordinated response where multiple units are involved.The detailed and updated 3D maps allow rescue teams to make informed decisions about where to focus their efforts, how to allocate resources, and when to deploy human personnel safely.

C. System Components of Cartographer Slam

LiDARs provide 360-degree scans around the robot, capturing detailed distance measurements necessary for creating accurate maps and detecting obstacles. These sensors are essential for navigating through rubble or cluttered environments typical in search and rescue scenarios. IMUs track the robot's orientation and acceleration, helping maintain its balance and accurate positioning when moving over uneven terrain or encountering sudden shifts in the environment. These sensors are used to track the movement of Señor Turtle over time, offering incremental updates to its position relative to a starting point, which is vital for path accuracy and correction of drift in the SLAM process. The figure 3 shows the cost scaling factor. The robot's optimal course is to go through an obstacle's center.

Processes real-time data from sensors to create submaps that represent the immediate environment. It ensures quick adaptation to changes, which is critical in dynamic rescue scenes where debris and obstacles may shift unexpectedly. Works to merge these submaps into a comprehensive and coherent global map. It also performs loop closure detection, which is crucial for correcting any drift or misalignment in the map due to the robot's movement over time.

D. Configuration for Rescue Missions

The system is configured to process data at a higher rate to ensure that changes in the environment are quickly reflected in the map. This is essential for navigating safely through unstable structures or rapidly changing conditions. Configurations are optimized to ensure that data from all sensors are accurately integrated, even under challenging conditions, such as varying lighting and physical obstructions common in rescue environments.

Routes and navigation plans are set to update dynamically in response to new data from the SLAM system. This enables Señor Turtle to adapt its path instantly to avoid newly detected obstacles or to approach newly identified areas of interest. Paths are calculated with safety margins that consider the physical dimensions of the robot and the typical debris sizes found in rescue sites, minimizing the risk of collisions.

Given the importance of map accuracy, loop closure algorithms are enhanced to recognize previously mapped areas

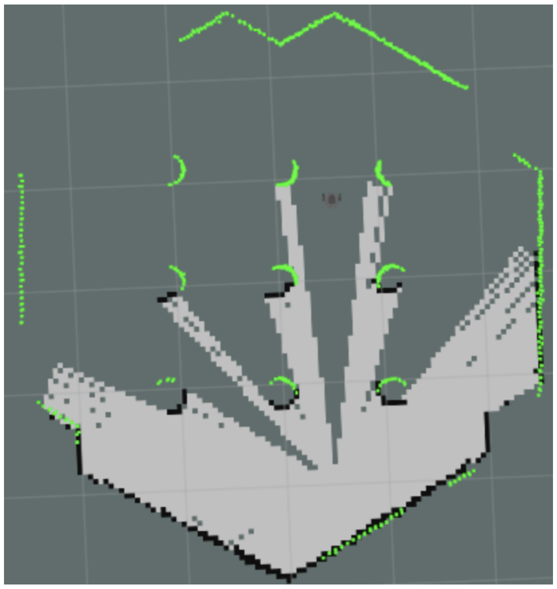


Fig. 4. Map Update Interval

quickly and efficiently, even under partial visibility or when significant map areas have been altered. The submap creation process is finely tuned to balance between detail and computational efficiency, ensuring that the robot can continue to operate for extended periods without excessive computational delays. The figure 4 shows the map update interval.

Configurations include robust error handling protocols to manage sensor malfunctions or data inconsistencies. This ensures that the system remains operational and can revert to safe states autonomously. In cases where SLAM data becomes unreliable, the system can switch to simpler navigation modes based on more direct sensor inputs, ensuring that Señor Turtle maintains basic functionality for retreat or manual rescue.

III. Local Planning and Control

A. Local Planning using Frontier Exploration

The system will use the map created by our SLAM packages to create goals based on unexplored "frontiers". These frontiers are the boundaries between the known and unknown areas of the map. The main goal of the algorithm is to group boundaries into a set of goals to explore then choose the frontier that maximizes information gain.

To keep the computational cost low, this implementation makes use of the simplicity of Breadth-First-Search to comb through the updating map. The algorithm looks for areas that lie within 50-80 'steps' of the current pose. A step will be a translation in the map equal to the map resolution. Once those goals are computed the algorithm chooses the goal closest to the robot and supplies that to a ROS topic.

B. Model Predictive Control

The system will use a modified version of Model Predictive Path Integral (m-MPPI) Control as its local path-planner. Given proper goals, the controller should drive the TurtleBot safely through the environment, avoiding any obstacles and reaching within a certain distance of the goal.

Standard MPPI control is a stochastic trajectory optimization algorithm that uses a dynamic model of the system to predict future states based on randomized actions. Then, the model costs each stored trajectory based on a custom cost function and uses weighted sampling to update the nominal trajectory.

For m-MPPI, the controller uses the following process.

1) Computing Rollouts: The algorithm samples a number of random angular velocities between a specified range. Then, using Unicycle dynamics (a differential-drive model) created by the esteemed Dr. Michael Everett for Northeastern's Mobile Robotics Course, trajectories are created following each angular velocity and a set linear velocity for a certain number of steps, or look-ahead states. Therefore, each trajectory will follow the same action and store the possible future states. In this, most of the stochasticity is removed from the original algorithm to cut computational costs as angle sampling occurs only once each loop.

2) Costing Rollouts: The custom cost function is simplified to check 2 features of each state in each trajectory; occupancy and distance to goal. The first cost added is the euclidean distance or norm between a future state and the given goal. The occupancy of future states can be calculated in one of two ways, the LiDAR scans or the SLAM map.

Initially, the SLAM map was used to check the occupancy of each future state. The pose of each state was transformed to map row and column indices and an occupied state received a cost of 1000 and an unoccupied state received a cost of 1. However, due to the map being updated as the robot explores, there would generally be missing occupancy data and the algorithm caused several collisions.

Therefore, the LiDAR scan was used to check occupancy. Numpy's norm and arctan2 functions were used to calculate the angle and distance between the current robot's pose and the future state pose. Then, the LiDAR range corresponding to the closest angle was used to determine whether the norm distance was within the acceptable distance of the detected obstacle. The occupancy costs remained the same at 1 and 1000.

3) Cost Evaluation and Control Execution: m-MPPI uses weighted sampling to get the nominal trajectory. As such, the probability of each trajectory is determined based on its cost. However, each loop, the next nominal action is not updated, but replaced with the new nominal angular velocity. Since each trajectory follows the same angular velocity and set linear velocity, the weighted sampling produces the nominal angular velocity to be used in the next action, not a nominal trajectory. Therefore, each action is independent of previous trajectories.

C. Tuning Hyper-parameters of m-MPPI

One of the many constants in life is the frustration of a robotics engineer while tuning hyper-parameters. One of the challenges faced in this project was tuning the controller to avoid obstacles. The hyper-parameters were tested in Gazebo environments (TurtleBot World and House) and initially, the TurtleBot was moving at full speed, sampling 30 trajectories, and had an angular velocity sampling range of 2*pi. However, while that produces several bloopers in our simulation, it would have caused enough collisions to damage the TurtleBot in a real life setting.

Therefore, through grueling trial and error, the following hyper-parameters were chosen for the controller:

- The number of trajectories to compute = 50
- The number of states computed for each trajectory = 10
- The set linear speed of TurtleBot = 0.07 (0.22 is the maximum)
- The range of sampled angular velocities = 2/3*pi
- The acceptable euclidean distance between a future state and LiDAR scan range to be considered unoccupied = 0.7

D. *Limitations of m-MPPI and Frontier Based Exploration*

As with all path-planning algorithms, m-MPPI comes with limitations. One common cause of failure is an unreachable goal. If the controller receives a goal that cannot be directly reached due to large obstacles or requires complex planning (like a maze environment), m-MPPI can get stuck. Therefore, the frontier exploration algorithm needs to be finely tuned to supply close goals that are directly reachable from the current state. Therefore, our BFS search algorithm does not go beyond 50-80 map steps. However, that limits the frontier exploration to shorter areas and can leave some areas unexplored. To avoid this, an additional local planner can be added to create sub-goals along the way to farther frontiers. In that setup, frontier-based exploration can provide global goals, no matter how far out they are, and the local planner would provide the local goals to the m-MPPI controller to get to those frontiers. Possible local planners include A* or Dijkstra's algorithm. This would be one of the next steps to implement.

Another limitation of the m-MPPI algorithm is the LiDAR scan resolution. If a thin enough obstacle presents itself, the scan might not pick up on it properly and the robot collides. One example is the table in the Gazebo House environment. The first few runs caused the TurtleBot to collide with the table legs and get stuck. One solution is to use a LiDAR with a higher resolution.

IV. Vision System

Our main goal with the vision system was to detect victims (represented by April tags) and get a good estimate of their position in the map. We utilized the apriltag_ros wrapper package by April robotics to detect april tags and publish their positions and ids over a /tag_detections topic. The inherent issue with the pose detection using a monocular camera (A raspberry pi camera in our case) is that it often gets the depth wrong, resulting in the apriltag showing up behind or in front of the wall as in figure 5. We explored the reasons why this problem occurs and we tried to rectify it.

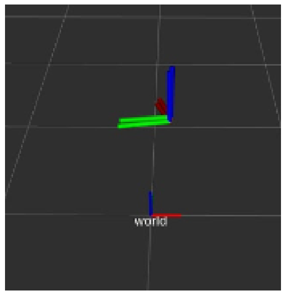


Fig. 5. Incorrect Pose estimate due to motion

A. *The Causes*

AprilTag, a robust visual fiducial system, is widely utilized for 3D pose estimation from a single image. However, when employing a monocular camera, the accuracy of pose estimation is compromised as the distance between the camera and the tag increases. This issue primarily arises from the projective transformation inherent in monocular vision systems, where image points are projected onto a 2D plane, causing distortions especially as the tag moves further away [9].

With increasing distance, the tag's image resolution decreases, reducing the number of pixels that represent the tag. This lower pixel density adversely affects the ability of detection algorithms to accurately localize the tag's corners and edges, which are crucial for precise pose estimation [7].

Additionally, lens distortions, particularly radial and tangential distortions, further exacerbate these errors. Such distortions skew the perceived position of the tag in the image, leading to inaccuracies in pose calculation, which worsen with increased tag distance and when the tag is positioned away from the center of the image [8].

$$\text{Pose Error} \propto \frac{1}{\text{Distance}^2} \tag{1}$$

The above equation (Equation 1) demonstrates the theoretical increase in pose estimation error as a function of the square of the distance from the camera to the AprilTag, under ideal imaging conditions. Practical challenges such as varying lighting conditions and partial occlusions can further impact the accuracy of pose estimation [9].

To mitigate these distortions, implementing strategies such as using higher resolution cameras, enhancing edge detection algorithms, or employing multi-camera systems for triangulation can substantially improve pose estimation accuracy, particularly at greater distances [6].

B. *Potential rectifications*

To address the challenges of distortion in pose estimation with AprilTags using monocular cameras, several effective strategies can be employed. One of the primary methods involves the integration of pose data into the 2D LIDAR frame through extrinsic calibration. This approach leverages the accuracy and range capabilities of LIDAR systems to enhance the robustness of pose estimation, particularly in complex environments [1].

The transformation of pose estimates from the camera frame to the LIDAR frame can compensate for the perspective distortions inherent in images captured from a distance. The calibrated system utilizes the depth information provided by LIDAR to refine the pose data obtained from the AprilTag detections, leading to more accurate and stable estimations [6].

Another technique we used recognizes previously identified tags and uses weighted averaging to update the new position, so the more you see the same tag, the more likely it is to become more accurate. The code snippet for this is shown below

```
if tag_id in self.closed_tags:
    print('UPDATING TAG:', tag_id)
    L = 0.95
    self.closed_tags[tag_id] = (1-L) * TMA
    + L * self.closed_tags[tag_id]
```

In addition to extrinsic calibration, other techniques can also significantly enhance the accuracy of pose estimation:

- **Higher Resolution Cameras:** Using cameras with higher resolution can improve the granularity at which AprilTags are captured, thus increasing the accuracy of edge and corner detection in the pose estimation algorithms [9].
- **Advanced Edge Detection Algorithms:** Employing sophisticated edge detection algorithms can mitigate the effects of pixelation and blurring that occur at greater distances, ensuring that the boundaries of AprilTags are more accurately recognized [5].
- **Multi-Camera Systems:** Implementing systems that utilize multiple cameras to capture different angles of the AprilTag can help in triangulating the tag's position more precisely, reducing errors introduced by angular and perspective distortions [3].
- **Machine Learning Models:** Machine learning models can be trained to recognize and correct for distortions in images of AprilTags, adapting dynamically to changes in distance and angle to improve pose estimation accuracy [4].

These strategies collectively contribute to minimizing the errors associated with pose estimation in environments where precise localization is crucial. Continuous improvements and integrations of these technologies are pivotal in advancing the reliability and applicability of monocular camera-based pose estimation systems.

C. Extrinsic Calibration

Extrinsic calibration of a monocular camera system with a LIDAR sensor is crucial for enhancing the accuracy of spatial measurements and pose estimation. This process involves determining the precise spatial relationship between the camera and the LIDAR device. In this project, a practical approach was adopted for calibration using manual point correspondence and computational techniques provided by OpenCV.

1) Manual Selection of Correspondence Points: The first step in the calibration process involved manually selecting 20 points in an image captured by the monocular camera. These points were carefully chosen to correspond to distinctive features that could be reliably identified in both the image and the LIDAR scan data. This manual selection is critical as it establishes a set of point correspondences between the camera's image space and the LIDAR's coordinate frame, which are used as input for the calibration algorithm. This process is illustrated in figure 6.

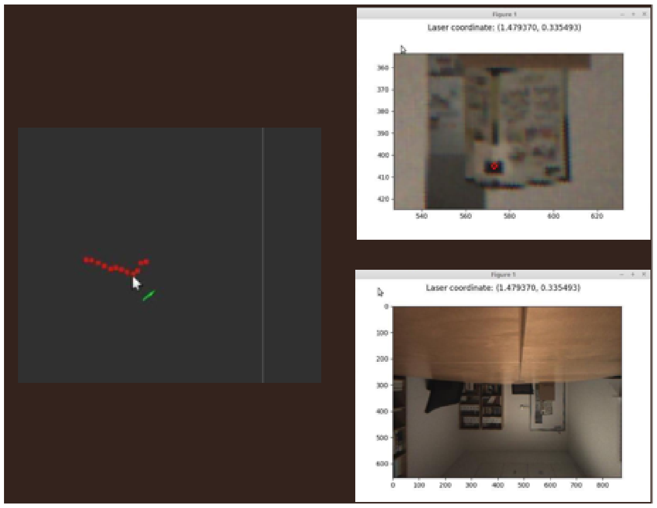
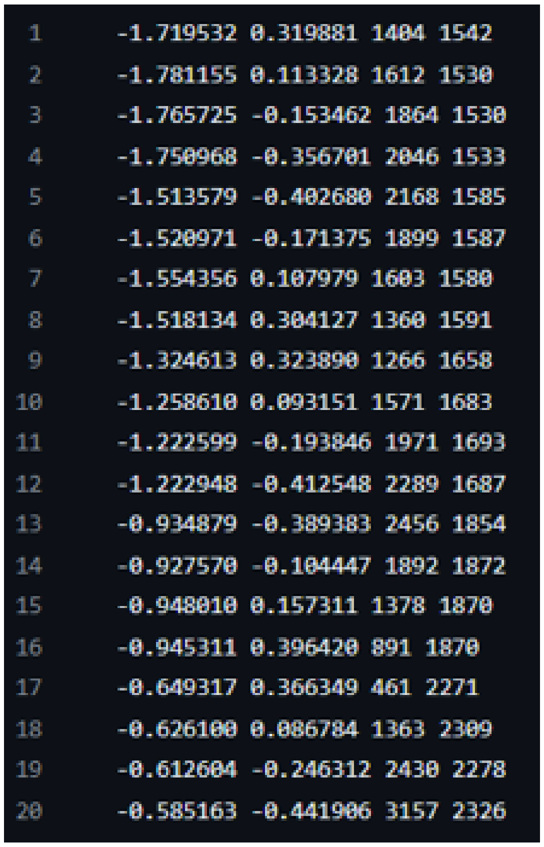

Laser coordinate: (1.479370, 0.335493)
Laser coordinate: (1.479370, 0.335493)

Fig. 6. Data Selection Process

2) Association with LIDAR Scan Points: Each of the selected image points was associated with corresponding points from the LIDAR scan. These associated points are treated as object points in the 3D space. The precision in selecting these correspondences directly influences the calibration's accuracy, as any error in point matching can lead to significant errors in the subsequent pose estimation. Figure 7 shows the data association saved in a text file in the format: Lidar x coordinate, Lidar y coordinate, Image x coordinate, Image y coordinate

3) OpenCV's SolvePnP: With the correspondences established, the next step involved utilizing OpenCV's `solvePnP` function. This function computes the pose of an object based on a set of 3D object points and their corresponding 2D projections in the image. The function uses the camera's intrinsic parameters and the established correspondences to compute the rotation vector and translation vector that describe the transformation from the 3D object points' frame to the camera's coordinate frame. The code snippet is provided below

```
imgp = np.array([imgp],dtype=np.float32)
objp = np.array([objp],dtype=np.float32)
```



```
-1.719532 0.319881 1404 1542
-1.781155 0.113328 1612 1530
-1.765725 -0.153462 1864 1530
-1.750968 -0.356701 2046 1533
-1.513579 -0.402680 2168 1585
-1.520971 -0.171375 1899 1587
-1.554356 0.107979 1603 1580
-1.518134 0.304127 1360 1591
-1.324613 0.323890 1266 1658
-1.258610 0.093151 1571 1683
-1.222599 -0.193846 1971 1693
-1.222948 -0.412548 2289 1687
-0.934879 -0.389383 2456 1854
-0.927570 -0.104447 1892 1872
-0.948010 0.157311 1378 1870
-0.945311 0.396420 891 1870
-0.649317 0.366349 461 2271
-0.626100 0.086784 1363 2309
-0.612604 -0.246312 2430 2278
-0.585163 -0.441906 3157 2326
```

Fig. 7. Data Association

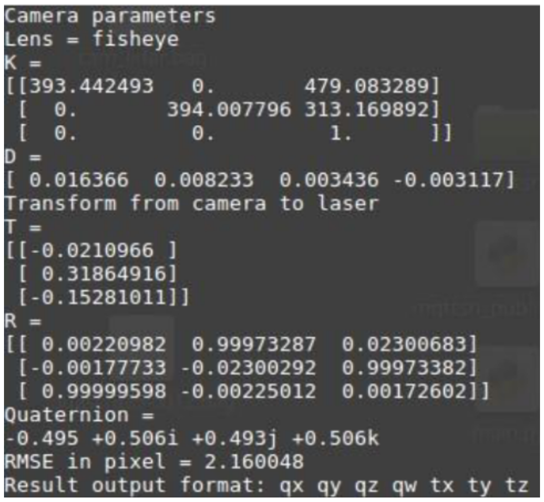
```
D_0 = np.array([0.0, 0.0, 0.0, 0.0])
retval, rvec, tvec = cv2.solvePnP(
    objp,imgp,K,D_0,
    flags = cv2.SOLVEPNP_ITERATIVE)
```

4) Rodrigues' Rotation Formula: To convert the rotation vector obtained from `solvePnP` into a usable rotation matrix, the Rodrigues' rotation formula was employed. This formula transforms the rotation vector into a full three-dimensional rotation matrix. The rotation matrix is particularly useful for integrating and interpreting the rotational aspects of the transformation in a more comprehensible form, facilitating further calculations involving 3D transformations.

```
rmat, jac = cv2.Rodrigues(rvec)
q = Quaternion(matrix=rmat)
```

5) Calibration Outcome: The outcome of this calibration process is a set of transformation parameters that define the spatial relationship between the camera and the LIDAR sensor. These parameters are essential for tasks that require merging data from both sensors, such as augmented reality applications, robotic navigation, and advanced driver-assistance systems (ADAS).

This method of extrinsic calibration, while manual and labor-intensive, provides a high degree of control over the selection of correspondence points, potentially leading to more accurate calibration results when performed meticulously.



```
Camera parameters
Lens = fisheye
K =
[[393.442493   0.       479.083289]
 [  0.       394.007796 313.169892]
 [  0.         0.         1.      ]]
D =
[ 0.016366  0.008233  0.003436 -0.003117]
Transform from camera to laser
T =
[[-0.0210966 ]
 [ 0.31864916]
 [-0.15281011]]
R =
[[ 0.00220982  0.99973287  0.02300683]
 [-0.00177733 -0.02300292  0.99973382]
 [ 0.99999598 -0.00225012  0.00172602]]
Quaternion =
-0.495 +0.506i +0.493j +0.506k
RMSE in pixel = 2.160048
Result output format: qx qy qz qw tx ty tz
```

Fig. 8. Final output of extrinsic calibration

Of course, it also has drawbacks, such as the fact that it won't work for drones that have roll and pitch, since all the correspondence is done with respect to 1 image resulting in a planar relationship. It works for our use case because the bot is expected to drive around on relatively flat surfaces.

V. Testing and Results

Initially, the project's main goal was to test the developed system in the real world using a custom Cardboard maze environment with April Tags QR codes to represent search and rescue victims. However, the team was faced with several challenges when prepping Senor Turtle for its journey.

When the TurtleBot was first received, initial investigations concluded that the battery was dead, the LiDAR wire connectors were broken, and some key wiring was missing from the setup. Once the new pieces were supplied and Senor Turtle was properly disassembled and re-wired, secondary investigations concluded that the SD card was missing. With no installed operating system or software architecture, the next steps were to buy an SD card and setup the architecture. Once the TurtleBot was fully setup and could be remotely controlled over Wifi, a new problem arose; the LiDAR stops spinning after bring up and does not relay information. This was investigated, and since the LiDAR spins properly when the bot is powered (but not brought up), the most probable cause was the LiDAR driver. Several attempts were made over several days to fix the hardware issue, however; due to time constraints, the team was not able to fix this issue in time and tested our work through Gazebo simulations only.

Therefore, the testing for the Search and Rescue algorithms occurred in 2 Gazebo environments, mainly the TurtleBot 'World' and TurtleBot 'House'. Since the Frontier Detection Planner was not fully developed at time of testing, the m-MPPI planner was fed goals manually in a queue. In these tests, the TurtleBot SLAM package with gmapping was used instead of cartographer SLAM due to computational costs. However, since m-MPPI was modified to only use the LiDAR scans for collisions, Cartographer SLAM was not necessary for these tests even though it does work. These tests aim to prove that m-MPPI is able to autonomously control Senor Turtle through the environments and avoid collisions while getting to the goals.

Please refer to the GitHub repository [10] for the m-MPPI ROS package and recordings of the controller's successes (and some bloopers). The results show that this paper's modified MPPI can work properly as a local planner at a lower computational cost.

VI. Teammate Contributions

Please note that all team members contributed to the presentation and report by writing up and presenting their respective work. All team members assisted others on their tasks as was needed.

1) Febin Wilson:

- Setup Cartographer SLAM in ROS and tested it on Simulations

2) Adnan Amir:

- Setup and tested Vision System (April Tag Detection)
- Developed Frontier Detection Algorithm
- Helped Setup TurtleBot

3) Ramez Mubarak:

- Setup TurtleBot physical system and OS/Software Configuration
- Developed and tested m-MPPI ROS package from scratch

References

[1] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-time loop closure in 2D LIDAR SLAM," in Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), Stockholm, Sweden, 2016, pp. 1271-1278.

[2] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Cartographer: Real-time simultaneous localization and mapping at scale," Google White Paper, [Online]. Available: https://opensource.google/projects/cartographer. [Accessed: Sept. 20, 2023].

[3] M. Y. Ziauddin, B. Larijani, H. Tawfik, and A. Al-Bayatti, "Advances in autonomous robotics systems: An analysis with special focus on contributions and applications of SLAM techniques," Journal of Engineering and Applied Sciences, vol. 10, no. 6, pp. 2554-2561, 2015.

[4] S. Thrun and M. Montemerlo, "The graph SLAM algorithm with applications to large-scale mapping of urban structures," International Journal of Robotics Research, vol. 25, no. 5-6, pp. 403-429, May-June 2006.

[5] J. Leonard, J. McDonald, M. Cox, C. Cadena, A. Durrant-Whyte, and B. J. A. Kruijff, "OpenSLAM: A collaborative approach to robotics research," in IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Tokyo, Japan, 2013, pp. 1552-1557.

[6] M. Sefidgar and R. Landry Jr, "Unstable Landing Platform Pose Estimation Based on Camera and Range Sensor Homogeneous Fusion (CRHF)," Drones, 6, 2022.

[7] R. Schouten, O. Arslan, and A. Isleyen, "Accuracy of Single Camera Pose Estimation with ArUco Fiducial Markers," 2021.

[8] D. Kriegman, E. Triendl, and T. O. Binford, "Stereo vision and navigation within buildings," 2005.

[9] J. Wang and E. Olson, "AprilTag 2: Efficient and robust fiducial detection," in Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2016.

[10] R. Mubarak, A. Amir, F. Wilson, "Mobile Robotics Final Project GitHub Repository"

GitHub Link:
https://github.com/RMubarak/Mobile-Robotics-Project.git