MIE443 Contest 1

Project Report

Team #: 27

Prepared By

Amin Heyrani Nobari (1003094821)
Farzaam Khorasani-Gerdehkouhi (1003011288)
Zarak Khan (1002056767)
Chandula Jayasinghe (1002912217)
Naib Gulam (1002197320)

February 13th, 2020

1. Problem Definition

For contest 1, the primary function is to program a TurtleBot 2 to autonomously explore an unknown maze after being placed in an unknown location within it. The maze environment is constrained to a boxed area of 4.87 x 4.87 m^2 and only contains static objects, an example of which is shown in figure 1. The static objects contained within the maze are restricted to cylindrical and rectangular shaped objects. However, the object size, quantity, position and orientation could be entirely different from that of figure 1. While the TurtleBot is exploring the maze, it must also completely map its environment and static objects using the ROS gmapping package, an example of which is shown in figure 2. Contest 1 also includes certain constraints that the team has to abide by which are listed below:

1. Must complete the exploration and mapping within 8 minutes.
2. Must incorporate a maximum TurtleBot linear speed of 0.25 m/s when navigating and 0.1 m/s when close to walls or obstacles to ensure consistency of maps between teams
3. Must be fully autonomous and not require any human intervention
4. Must use feedback from sensors to conduct the mapping and cannot use fixed sequences within the code

Figure 1: An example of a mapping area

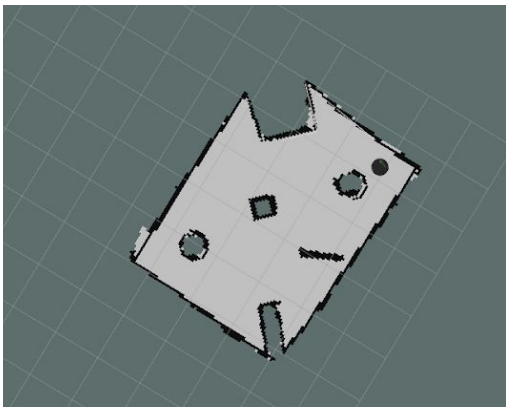

Figure 2: An example of a created map

1.1. ROS TurtleBot

The TurtleBot is built on open-hardware and open-source software allowing easy customization for any desired robotics application. The TurtleBot Software Development Kit is built upon the widely used TurtleBot Operating System (ROS) and is equipped with a Xbox 360 Kinect sensor and a Lenovo 11e laptop. The two components are placed on an iClebo Kobuki mobile base, on which 3 supporting plates are stacked as shown in figure 3.

The Kinect sensor has a RGB camera for object detection and recognition, depth sensor for distance measurements, and a microphone array that can locate locations of sounds as shown in figure 4. Moreover, the TurtleBot's Kobuki mobile base components includes an odometer for linear and angular motion and incorporates 3 bumpers that can be detected when pressed.

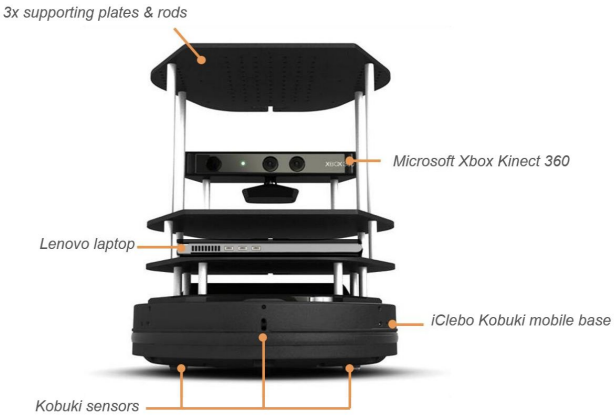

Figure 3: The Turtlebot system used by the team

Figure 4: The Xbox 360 Kinect device

2. Strategy and Motivation

During the preliminary stages, the team discussed different strategies to efficiently map the maze. From these discussions, two different methods of maze navigation were considered: wall following and frontier search methods.

2.1. Wall Following vs. Frontier Search

Wall following refers to the motion of the TurtleBot where it continuously tracks and traverses along a wall. The user has the option of either traversing along the left or right sides of the wall. This requires the TurtleBot to be in close proximity to the wall being tracked at all times. The team produced an algorithm which allows the TurtleBot to pursue a wall on its right side. If the Turtlebot loses sight of the right wall, a sequence of events is initiated to turn the TurtleBot in a clockwise fashion until a right wall is spotted. Once found, the TurtleBot tracks the wall while being a certain distance away from it to avoid collision. Additionally, if the TurtleBot's front is blocked by any obstacle, it takes a left turn to keep the wall on its right side. Although wall following is a great way to traverse through the maze and map out the environment, the team opted out of this method. This is because the TurtleBot needs to constantly keep track of the right wall and in any event if the TurtleBot drifts away or comes too close to the wall, it would have to adjust itself to be parallel to the wall, which is time consuming. Furthermore, the TurtleBot behaves erratically, when it is attempting to follow a cylindrical object. Since the Turtlebot is always in pursuit of a right wall, if placed near a wall, it will only trace the boundaries of the maze without entering the middle region.

As a result, the team decided to implement a method that is a sort of hybrid between frontier search and wall following based on navigation with a grid system. In this method, the TurtleBot starts off in the first available unblocked direction, following a set of instructions to traverse through the maze, while avoiding obstacles and adjusting direction of motion based on the availability of free paths. Since there is a grid system in place, the algorithm keeps track of all the areas the TurtleBot has visited. In any event that the

TurtleBot revisits a grid, it makes sure it goes in an available path other than the ones previously taken. This ensures the TurtleBot does not repeatedly loop around an object. Unlike wall following, this algorithm for search does not waste time adjusting itself to a specified wall, rather it uses that time to carry out scans to map out the environment. Overall, this method performs mostly superior to wall following, but it comes with a small disadvantage. Since this algorithm is heavily based on the availability of free paths, the TurtleBot might have trouble entering areas with small entrances, while also adding the possibility of for the robot to still get stuck in a loop if it cannot find any unexplored or available paths. This can, however, be resolved by employing a proper set of instructions to supplement the primary algorithm.

The claims that are made in the above paragraphs were all verified with testing both alternatives and comparing the results of the gmapping. These experiments and simulations were the primary driving force in the decision making process, when it came to picking the aforementioned alternative.

2.2. Strategy for Random Navigation

The following outlines all the functions and abilities the TurtleBot needs to possess in order to attain the required objectives. These are implemented through sensor utilization and controller design which will be dealt with in great detail in the following sections. The overview of the strategies that were used as the basis of developing each algorithm are broadly discussed below:

- The TurtleBot should be able to perform translational and rotational movements such as advancing forward/backwards and rotating clockwise/counter clockwise. Different combinations of these basic movements will allow the TurtleBot to navigate through the maze.
- The TurtleBot should be able to detect its surroundings. This is achieved through the use of the Microsoft Kinect depth sensor.
- Based on its sensed environment, the TurtleBot should be able to make decisions as to which path to adopt after its initial placement in the maze.
- The TurtleBot should be able to perform obstacle avoidance based on its sensed environment. It should slow down when it is within a certain distance away from an object. Once it is in close proximity to an object, it should halt and turn in an appropriate direction to adopt the most efficient path.
- The TurtleBot should also avoid constant loops around objects, which is achieved through the utilization of the grid system.
- The Microsoft Kinect depth sensor outputs faulty readings when the distance is less than 0.4 m. Therefore, the Turtlebot should be able to detect obstacles when the depth sensor exhibits erratic behaviour. This can be achieved by utilizing the bumper sensors installed at the base of the robot.
- The Turtlebot should map its surroundings using gmapping and record the data as it traverses through the maze. It should perform routine sweeps to ensure the creation of a thorough and accurate map.

3. Design and Implementation

Now that the primary design strategies and alternatives are discussed and the choice of alternative is made clear, in this section the details of the final design and implementation of the aforementioned algorithm will be discussed.

3.1. Sensory Design

This section details the sensors chosen by the team as well as the motivation behind why we chose to use them to fulfill contest 1's problem definition.

3.1.1. Available Sensors

A variety of sensors were available for use by the team. These were built into two distinct components of the Turtlebot: the Kinect sensor and the iClebo Kobuki mobile base.

3.1.1.1. Kinect Sensors

The Kinect system contains all of the remote sensors available for use. The following diagram indicates the sensors in the system as well as their positions.

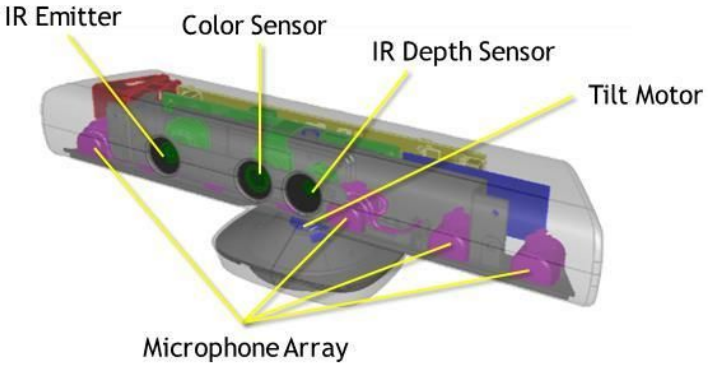

Figure 5: Sensor positions on the Kinect system

Kinect Depth Sensor: The depth sensor utilizes an infrared projector coupled to a monochrome CMOS sensor to create 3-D imagery of the surrounding. The distances to objects in the projected area are measured by the transmission of a speckle pattern of infrared (IR) beams by an IR laser emitter, which is, in turn, detected by an IR sensor upon reflection from the object. The object deforms the pattern by virtue of its geometry. The deformed pattern received back is then compared with the original pattern to determine the depth of objects in the environment.

RGB Camera: This camera makes use of red, green, and blue channels to take color images of the sensor surroundings. As these colors are the additive primary colors, any other color can be recreated using a combination of the three. Each pixel in an image is given separate red, green, and blue brightness values and these are combined to create its overall color. As such, a camera of this sort will provide more realistic and informative images of its surroundings than an equivalent grayscale or black-and-white camera.
Microphone Array: The Kinect microphone array can be used to pick up and locate sounds and sound sources in the surrounding environment. If given enough information, it can also

identify specific sounds when confronted with a wide array of different noises. To perform effectively, it consists of four separate microphones arranged together.

3.1.1.2. iClebo Kobuki Mobile Base Sensors

The base of the Turtlebot contained the physical and analog sensors and systems available. The following diagram highlights a number of the important sensors present.

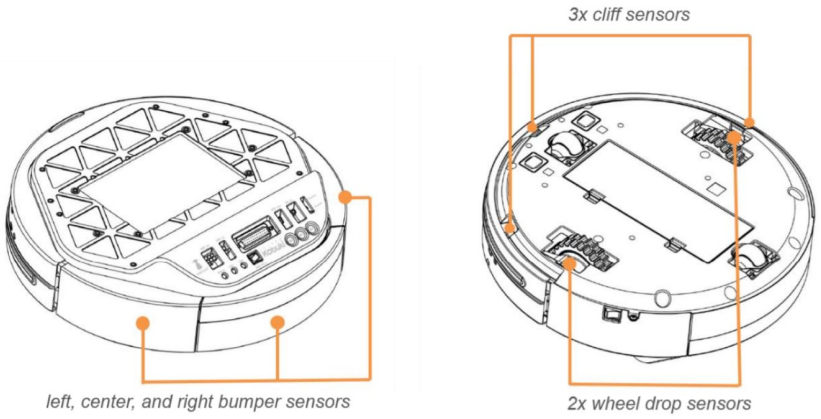

Figure 6: Sensors on the mobile base

Bumpers: The TurtleBot contains three bumper sensors located at the front, left and right of the base of the TurtleBot. Each of these sensors is triggered when the respective portion of the base collides with an obstacle.

Wheel Drop Sensors: These sensors are used to detect floor transitions and drops, helping the base navigate through difficult terrain that cannot be detected by the Kinect sensors. They work by considering either the elevation or the ground contact of the base wheels [1]. The mobile base used by the team contains two of these sensors, one on either side.

Cliff Sensors: These systems are usually infrared radiation emitters that make use of the concept used by the depth sensor discussed beforehand. They, however, point directly down towards the floor underneath the base. They emit IR signals and detect their reflections. If a signal isn't reflected back, the base assumes that there is a large "cliff" or vertical drop ahead and avoids that area [2]. The iClebo Kobuki base contains three of these sensors, one underneath each bumper.

Gyroscope: The 1-axis gyroscope is used to measure the angular orientation (yaw) and the angular velocity of the TurtleBot. As such, it is very useful in providing information about the orientation of the robot.

3.1.2. Sensor Choices

Given all of the sensors available, the team chose those most suitable for use in this contest. These were the depth sensor from the Kinect, and the bumpers and gyroscope from the iClebo Kobuki mobile base. Each of these sensors had a set of roles to fulfill the problem statement and the reason for neglecting the other sensors will also be discussed.

The depth sensor was chosen due to a need to be able to quickly identify obstacles remotely in order to succeed at this contest. It provided a proven and effective method to identify obstacles and blockages using IR radiation. If the other alternative, the RGB camera, was chosen for this task, the identification of obstacles and their distances would have been exceedingly difficult. This would be due to the fact that any images taken by the camera would need to be analyzed and processed using complex computer vision models. This would require large amounts of processing power and/or time, adding needless difficulty and complexity to the mapping process.

The bumpers were used because they were the only viable collision detectors on the Turtlebot. The cliff and wheel drop sensors were only capable of identifying elevation changes and the gyroscope could only be used to detect changes in the orientation of the robot. None of these were capable of detecting collisions, a feature that was deemed necessary by the team in order to ensure the Turtlebot could navigate the contest map without getting stuck.

A similar situation was observed with the gyroscope as it was the only sensor capable of providing useful odometry information regarding the orientation and position of the Turtlebot. None of the other sensors, including those on the Kinect could provide the type of data that the gyroscope could, making it the only suitable choice.

3.1.2.1. Depth Sensor

The depth sensor is the primary means by which the Turtlebot can 'see' its environment. The bumpers can also indicate obstacles, but at too late a time to actually avoid them. Information from the depth sensor plays a role in both navigation and mapping.

Navigation: The information obtained from the depth sensor is utilized in the 'laserCallback' function, which determines the minimum distance in the laser scan for a 25° field of view. The field of view was limited to 25° from the 60° available because that was found to be a sufficient enough field of view to detect objects ahead while disregarding readings on the far peripheries which would have no effect or negative effects. This specific number was determined experimentally, by trials with different fields of view. Additionally, the function also evaluates the minimum distances in [-30°, -17.5°] and [17.5°, 30°] field of view, which provides information about objects located on the slight right and left of the TurtleBot. This is depicted in figure 7. Based on this distance data, the TurtleBot avoids obstacles in the maze during navigation for a faster and more smooth motion. The sensors in their entirety are also used to find the minimum measured distance in any direction, which will be used to slow down when objects are closer to the robot in order to prevent hard collisions.

Mapping: During navigation, gmapping is activated, which utilizes depth sensor information from the entire horizontal viewing angle (57°) to create a 2D map of the unknown environment. The software RViz provides a visual representation of the mapped maze.

3.1.2.2. Bumpers

If an obstacle is not detected by the depth sensor, the bumpers are used by the Turtlebot to understand that a certain path is blocked. As such, they too play a role in navigation. However, they also have safety implications, keeping the Turtlebot from harming itself and hitting dangerous obstacles.

Navigation: If the front bumper is pressed, the TurtleBot moves backwards. If the left bumper is pressed, the TurtleBot rotates clockwise. Finally, if the right bumper is pressed, the TurtleBot rotates counterclockwise. However, these movements are only performed for 1.5 seconds.

Safety: Since the kinect depth sensor can only sense distances greater than 0.4 m in the desired field of view of 25° , the bumpers serve as a safety net in instances where the depth sensor fails to detect an obstacle in front of it. With the help of these bumpers, the TurtleBot can follow a sequence of specific instructions outlined above.

3.1.2.3. Odometry Information (Gyroscope)

The other set of sensory data that were utilized in the control system was the odometry data, which were provided directly from the robot based on the motion of the robot. This set of data includes the position and orientation of the robot, which is primarily what is used to update a grid data, as this is the basis of how it is determined if a grid is visited or where the robot is currently.

3.1.3. Sensory Implementation

A number of functions were mentioned in the sensor procedures of the previous section. These allow for easy communication between the Turtlebot hardware and a controller. Each allows for the exchange of a set of particular information between the robot and controller.

- The Odom Callback function communicates the X and Y positions, and the yaw angle.
- The Bumper Callback function communicates any bumper presses.
- The Laser Callback function communicates laser distances for the depth sensor.

3.1.3.1. Odom Callback Function

This function subscribes to the odom topic that constantly publishes updated messages from the gyroscope and the encoders, providing information about the orientation and the X & Y position of the Turtlebot. The data is fed into posX, posY and yaw (converted from radians to degrees) variables, which are utilized for robot control, as mentioned in later sections.

3.1.3.2. Bumper Callback Function

This function subscribes to the bumper topic which constantly publishes messages from the bumpers, providing information about bumper states. If any of the bumpers is pressed, bumper flags are updated with new states which are then fed into this function. The states of each of the three bumpers (namely left, right and center) are stored in variables. These are then used to perform specific bumper maneuvers based on which bumper is triggered and is outlined in detail later in the report.

3.1.3.3. Laser Callback Function

This function subscribes to the laser topic which constantly publishes messages from the IR depth sensor, providing information about the minimum distance from an object. The minimum distance is obtained from both the 25° field of view and from all the laser sensors. These values are stored in variables containing these minimum distances and is responsible

for slowing down the robot when near a wall or obstacles and local traversal of the TurtleBot

3.1.3.4. Sensor Usage Visualisation

The following diagrams illustrate how sensory data is used by the controller algorithm as well as the general flow of sensory information.

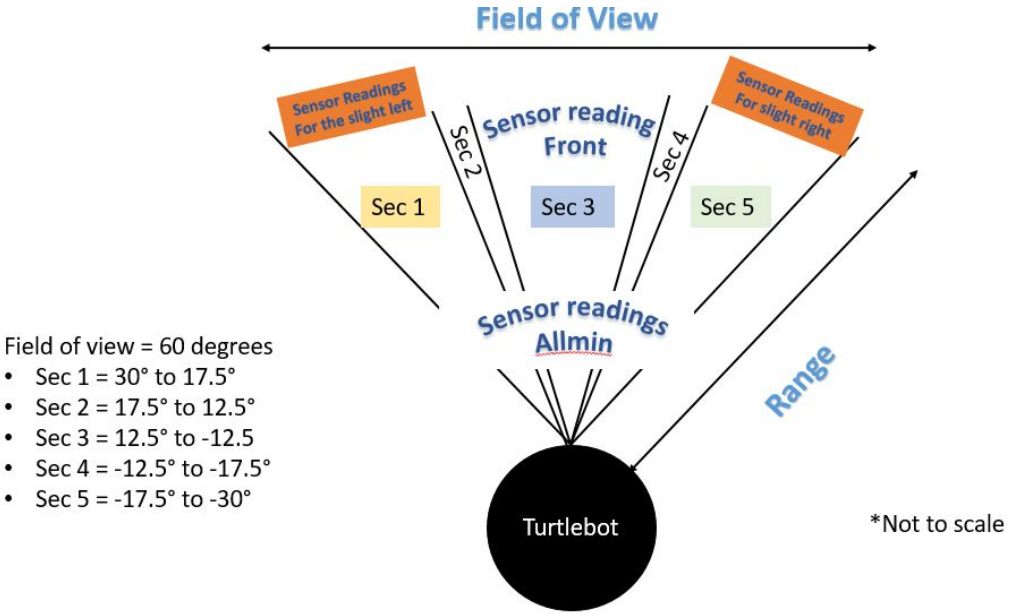

Figure 7: Turtlebot viewing range and response instructions

3.2. Controller Design

The implementation of the aforementioned strategy for this contest, including the choice of controller approach and the multiple algorithms and monitoring tools developed for this project are discussed in detail in this section. Several references are made to the C++ code developed for this project, which can be found under appendix A. It is important that the broad control approach and design choices be discussed to lay the foundations for a more comprehensive understanding of the detailed implementations of the control system, for this reason the type of control architecture and the general control approach is going to be discussed first:

3.2.1. Type of Control Architecture

First and most obvious choice that was made in this project was to choose an intelligent control system over a sequential one, as the task of exploring and understanding an environment that is completely unknown to the TurtleBot cannot be done through any type of sequential steps. As a result, the control approach is mostly limited to an intelligent system, as any sequential approach will most likely be limited to a reactive architecture, which in turn would probably not explore the environment successfully.

Given the intelligent approach for this project, the type of control architecture that was picked and developed was one that involved a *hybrid control* approach between reactive and deliberate control. In this approach the primary guiding force behind the decisions the TurtleBot makes is it's sensor readings from the environment, meaning that the control

system implemented primarily *reacts* to the environment when needed, however simple reaction to obstacles and paths is not sufficient for successful exploration of the environment. This is because a simple sequential and repetitive decision making approach would likely leave the TurtleBot moving around in loop or repeating a certain pattern of motion that is not guaranteed to explore the environment completely, hence introducing the possibility of an incomplete map of the environment and failure to fulfil the contest's primary goal. Therefore, a simplified perception of the environment is also developed as the TurtleBot moves around the environment, which is then used in any future decision making to augment the reactive control and introduce some *deliberation* to the system, in order to ensure a more comprehensive exploration of the environment. In this approach, the TurtleBot actively improves its perception of the unknown environment as new information from the sensor, and odometry data is provided to the TurtleBot. The details of how these control systems are implemented are discussed further in the sections following this one.

3.2.2. Control System Details

Now that the primary architecture of the control system is understood, the approach in implementing such a control system must be discussed before the more low level information regarding the specific algorithms can be dissected. As mentioned earlier, the control system is one that is hybrid and is primarily reactive, but also deliberate. Therefore, several aspects of the control system are working at once to produce decisions for the TurtleBot. The system can be best described through the flowchart in figure 9, where the TurtleBot's control system is described visually:

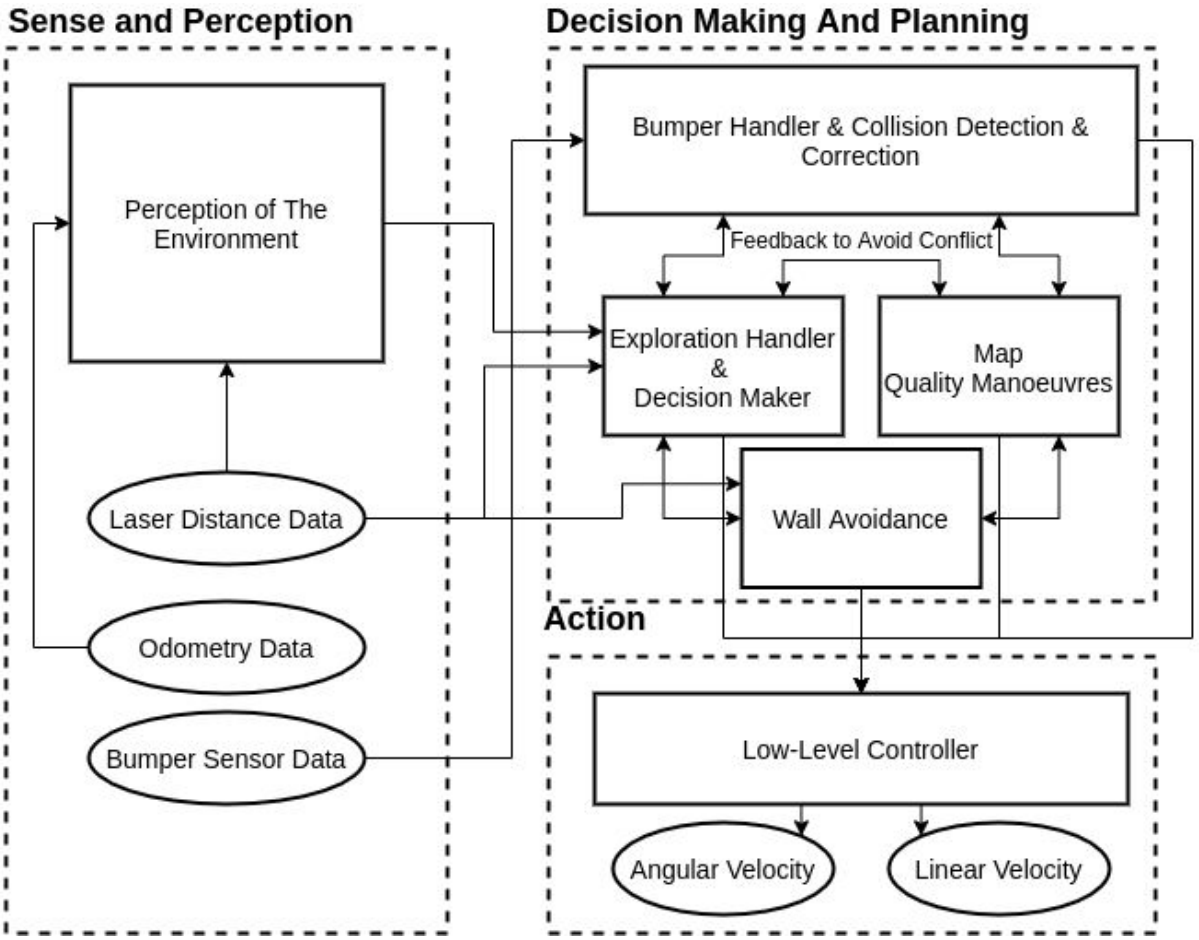

Figure 8: Control System High-level Diagram

From here the main features of the control system can be described. The control system constantly updates the TurtleBot's perception of the environment based on laser sensor data, when appropriate, to identify blocked areas of the environment, while at the same time the odometry data is used to also augment this perception based on the areas that the TurtleBot has visited, while also keeping track of how many times certain areas have been visited. Finally the bumper data is also kept track of to detect collisions and steer clear of obstacles too close for the TurtleBot to detect when handling explorations. These are basically the sense and perception of the TurtleBot and its control system.

The other part of the control system that makes the decisions and plans the motion of the TurtleBot uses the sense and perception discussed prior to come up with the appropriate angular and linear velocities that are then sent to the TurtleBot directly through the low-level controllers, which are primarily the motor drivers and possibly PIDs that are prebuilt into the TurtleBot 2. In this planning stage, four main elements are implemented. These elements each have priorities which determine how conflicting decisions are handled if different elements of decision making ask for contradicting outputs. In this system the highest priority is given to the bumper handler, as if the bumper is triggered there is a collision, which is a critical issue that must be resolved prior to any further action. After this the priority goes to the exploration handler, which uses the laser data and perception of the environment to make decisions with regard to how or if the TurtleBot should react at every moment. The third element which performs scan maneuvers that improve the quality of the gmapping results are given the lowest priority since these maneuvers are simply not

essential, and will only be allowed to occur at specific moments, where the scan does not interfere with rest of the TurtleBot operations. Finally, the last element is the wall avoidance which is not prioritised specifically, as it is an augmentation control system, which augments the final decision based on the laser data from the left and right to avoid collisions or guid the TurtleBot smoothly around obstacles, therefore it is only not active during bumper handling, since again bumper handler is given complete priority in the system.

This overview even though complete in describing the behaviour of the TurtleBot at a high-level, do not provide any insight into the nuances and intricacies of the actual implementation of the control system, as the implemented C++ code to achieve the aforementioned control system involves specific details that must be taken into account for practical reasons, however before getting into the code and algorithm itself, the specifics of the algorithms and implementations of the control system must be described, without going into the code. These explanations are provided in three subsections:

3.2.2.1. Description of Perception

As mentioned before, the TurtleBot uses the laser sensor data to build a perception of its surroundings, and later uses this perception to make decisions. This perception is built around a grid system, which has a limited resolution. The reason for implementing a low resolution is that the grid system is updated and developed based on the odometry data from the TurtleBot. The odometry data from the TurtleBot is not very accurate and as time passes the error in the odometry data accumulates which makes the odometry data less reliable, therefore if a fine grid system was implemented the perception would not remain consistent during exploration, and essentially make the grid system, hence perception useless for decision making. However, if the grid system remained coarse the effects of the error would be less notable and the grid data would be useful for decision making. This rough grid system in essence is monitoring certain areas relative to the TurtleBot location rather than exact coordinates of where the TurtleBot has been.

The grid system is implemented so that it can keep track of two main aspects, the TurtleBot motion and sensor data. The TurtleBot motion is kept track of by marking certain grids(areas) marked as visited, the system also records the TurtleBot's motion in each area by keeping track of not only if the area has been visited, but also how many times the area has been visited. The second aspect of the grid as mentioned uses the sensor data. The sensor data is used to mark certain grids(areas) as blocked, meaning that that area is most likely unreachable and the TurtleBot will try to avoid those locations if it comes across them at any point in the future. To understand this better figure 10 is presented below which is retrieved from one of the trial runs of the TurtleBot.

As it can be seen in the figure some parts of the perception grid system are marked dark gray, which is indicative of the fact that those areas are not visited therefore unknown and/or unexplorable, these grids(areas) are what are favoured by the exploration handler, which is discussed in detail later. The grids marked black, are areas that during exploration have been noticed to be blocked using the laser sensor data. Finally, other areas are areas that have been visited and considered known areas. Another important thing to note here is that as it can be observed some visited grids(areas) are brighter than others. The brightness of each grid is indicative of how many times each grid has been visited, which as mentioned before is being kept track of using the odometry data. This grid system is what is the basis of

most decision making in the control system, which is such that the exploration handler attempts to move towards unexplored regions(grids) to cover the most area possible. This system is implemented to allow for some deliberation in the control system and introduce a systematic means of making decision that are not merely random or reactive, but are working towards achieving the goal of the project which is to cover the majority of the map and in turn provide data to the gmapping algorithm to build a useful and accurate map of the environment.

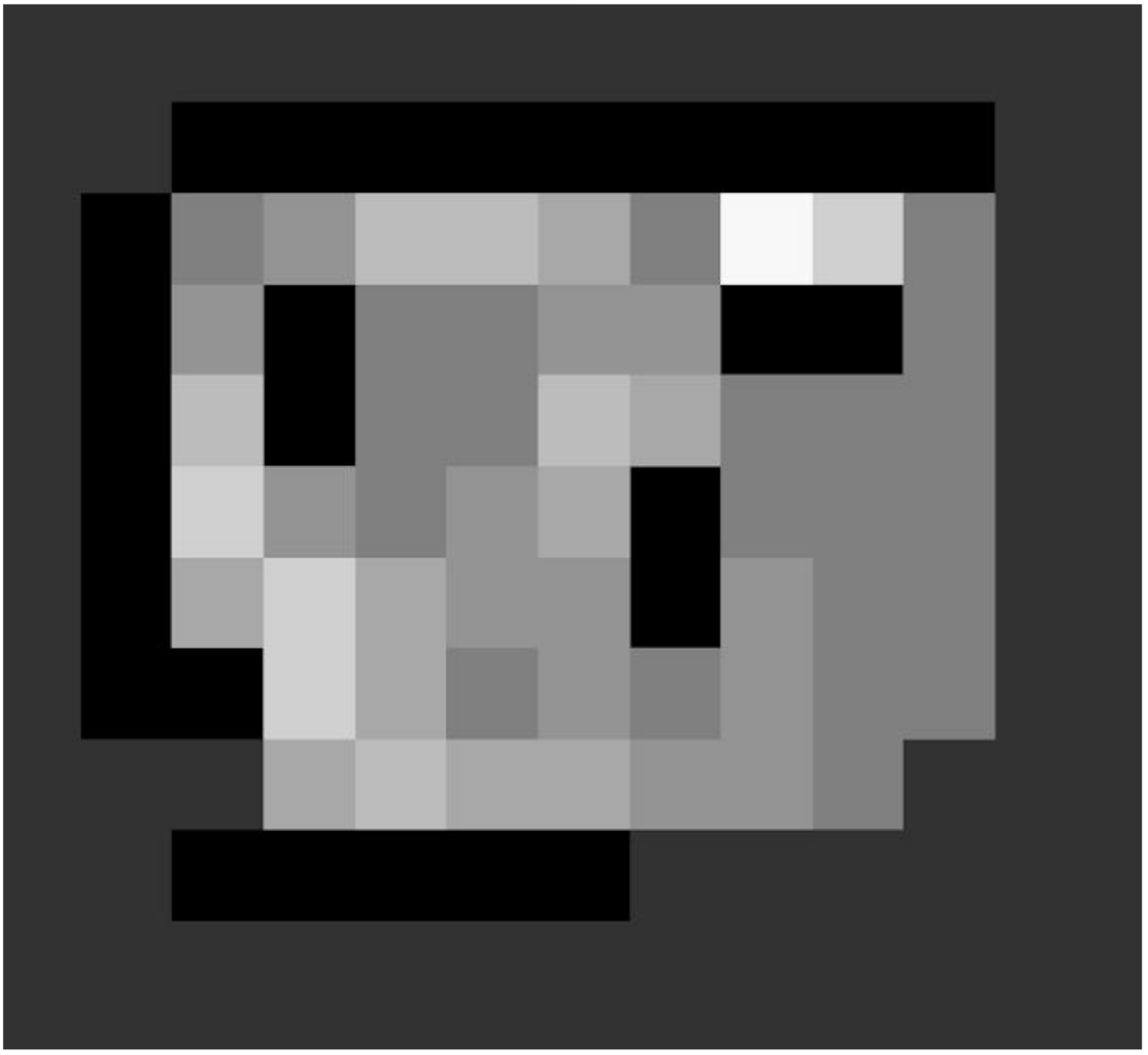

Figure 9: Example of A Grid System At The End of One Trial Run

3.2.2.2. Decision making and planning

The decision making process includes four main elements. They are bumper handler, exploration handler, mapping quality maneuvers(scanning), and wall avoidance. The bumper handler which has the highest priority is triggered if any of the bumper switches are triggered. When this happens all operations are reset and put on hold until the bumper maneuver is finished. The bumper maneuver is performed based on which bumper switch was triggered. If the center switch is triggered the maneuver will simply be a backwards motion for 2 seconds. If the right or left switches are triggered however, the maneuver will

be notably different. The maneuver will involve a backwards motion for 1.5 seconds followed by a backwards turn away from the direction of collision for 0.5 seconds and finally a 1 second of forward motion. This is best understood by looking at the following figure:

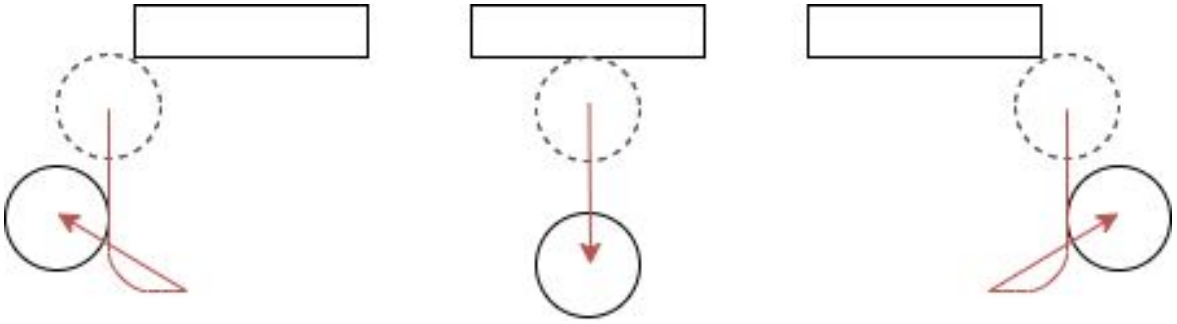

Figure 10: Bumper Maneuvers

During these bumper maneuvers nothing else is allowed to take place and once the maneuver has been completed normal operation will be continued. In this way it can be said that the bumper maneuver is analogous to timer interrupts, and is performed asynchronously and the control system is allowed to continue after the asynchronous operation has been finished.

The next part of the planning is the exploration handler. This aspect of the control system is an approach inspired by the nearest neighbor approach combined with other considerations. This is because the TurtleBot will only look at the nearest grids to its current grid and will not take into account grids further away from it. This part of the control system is implemented as such:

- At any given moment the minimum distance to obstacles is measured at the front , and if the distance is lower than a set threshold the TurtleBot linear velocity will be set to zero regardless of grid values(reactive).
- In case of the TurtleBot not being blocked by anything in front of it(25° of sensor data at the front of the TurtleBot), the TurtleBot will continue only if the grid that the TurtleBot is facing is unexplored or if all eight of the neighbouring grids are explored, or if the TurtleBot is on a grid that has been visited 3 or more times(this implies that the TurtleBot is stuck in a location which means that the TurtleBot should priorities getting out of that area so stopping would be counterproductive).
- In any of the cases above that the TurtleBot does not continue forward it will go into search mode until one of the above conditions are met or the TurtleBot has no alternative that meets those conditions in which case the TurtleBot will continue in the first direction available to it.
- During a search mode the TurtleBot's linear motion is stopped and it will begin to rotate to the right until the aforementioned conditions are met or a full rotation(full search occurs). In the case of full search the TurtleBot will simply move in any available direction. It is important to mention that if only one unknown neighbor exists the direction of rotation will be such that it reaches the direction facing the only unknown grid faster, be it right or left.
- If the robot is stuck at any grid for more than 20 seconds all the grids adjacent to the current grid is set to blocked as this is practically the reason for this happening. This will force the robot to stop searching and move in the first available direction.

- Finally, if at anypoint a grid is visited 4 times the default rotation direction will be flipped(from right to left). This is because such a behavior implies the robot is stuck in a loop which will need the direction of motion to be changed to allow for the robot to get out of a repeat pattern.

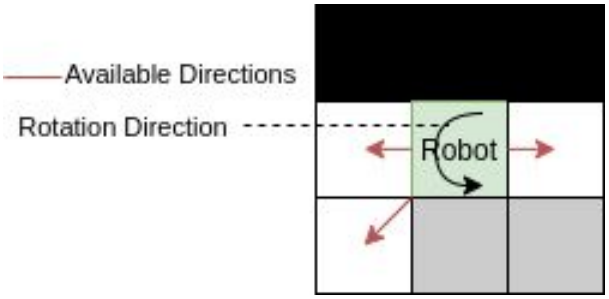

Figure 11: Graphical Representation of Search Mode

The third aspect of the planning, is the map quality maneuvers, which is a 360° scan with slow angular velocity, the overview of this aspect is as follows:

- No scan is done during any operation, whether it be bumper maneuver or exploration.
- No scan is done until at least 2.5 minutes have passed, this is because without any gmapping data produced first an early scan significantly lowers the gmapping quality.
- After 2.5 minutes an initial scan is made, the position of the scan and its time will be saved onto a vector variable.
- At any time a scan will be triggered if the distance from the previous scan is more than or equal to 1.8m, or 2.5 minutes or more has passed from the last scan.
- During a scan all other operations are put on hold until the scan has been completed.

As evident the scanning is similar to the bumper handler in that once triggered no other operation is allowed to happen until scanning has been completed.

The final aspect of the planning element of the control system is the wall avoidance system. This system works by looking at 12.5° of laser data from the left and the same amount from the right. If the value of either is less than a certain threshold some angular velocity will be added after a decision has been made, to steer away from the walls and obstacles on the sides of the TurtleBot that will not be caught by the front sensors on the TurtleBot. It is also important to note that if both the left and the right side sensors are reading values **lower or near** the threshold the adjustment will not be made as the TurtleBot is too close on both side indicating that the TurtleBot is passing through a tight path, which means that these corrections will simply not be helpful as they will most likely lead to the TurtleBot shaking to the left and right continuously.

3.2.2.3. Low-Level Controllers and Global Adjustments

The last stage of the control system is the action stage which is primarily the low-level controller. However, before the primary controller is explained it is important to mention the global adjustment controller in the system. This global adjustment controller changes the default linear velocity which will be used for all linear motions. This global linear velocity is set based on laser distance values. For this purpose all the laser sensor values are measured and the global linear velocity value is set based on the minimum distance read in

any direction. Based on this minimum value if at any direction the distance is between 0.8m and 0.55m the global linear velocity will be set to 0.15m/s, and if this value is below 0.55m the global linear velocity is set to 0.1m/s, finally in any other case the velocity will be set to a maximum of 0.2m/s. The velocity values are mainly based on the contest constraints, which require the velocity to be lowered to 0.1m/s near obstacles and never exceed 0.2 m/s.

The other part of this stage is the low-level controller, which in the case of the TurtleBot, would be the motor controllers and the PID that might be present in the motor drivers to achieve the required velocities properly.

3.2.3. Control System Implementation (Code & Algorithms)

Now that the overview and high-level details of the control system are described, the details of the algorithm and code to implement the algorithms must be discussed. For this purpose, first all the functions and variables inside the code will be dissected and later the complete functionality of the entire code will be explained. In this section certain parts of the code are explained which can be found under appendix A.

3.2.3.1. Functions

The full code contains multiple functions that are used to implement the aforementioned control system. Each function will be discussed in detail and any algorithm implemented within them will also be dissected in this section.

The first set of functions that must be discussed are the callback functions that are triggered every time new data is available from the sensors on the robot. These functions primarily handle retrieving and processing sensor data, however some of the control functionality is also done inside these functions. Since only three sets of sensory information are being used there are three callbacks corresponding to the three sets of sensory data. The three callback functions are laserCallback, bumperCallback, odomCallback, which operate as explained below:

- **LaserCallback:** This function has 2 main parts.
    - The first part calculates the minimum distance in the 25° field of view in front of the robot, 12.5° degree field of view on the leftmost and rightmost angles available through the laser sensors, and finally the minimum distance in all measured laser sensors. These numbers are then checked to ensure the laser readings exist and if laser readings do not exist, which might happen when obstacles are too close, the values of minimum distance are set to zero.
    - After the distances are determined, this call back is also responsible for the global speed adjustments, which it will do using the minimum distance value in all laser sensors, and lower the speed of the robot when obstacles are close to the robot.
- **BumperCallback:** This function has a very simple functionality, it saves the state of the bumpers in a global variable, and after this sets a flag that indicates whether or not any bumper was pressed.
- **OdomCallback:** This function takes the odometry information and saves the position and yaw information and determines the grid the robot is currently at, and the grid direction that the robot is facing, this is done using the get_grid_direction function, which is explained later. This callback function then calls the updateGrid function which updates the robot perception(grid values) based on the odometry data. This

function also activates or deactivates an all_known boolean flag that signals whether the neighbouring grids are all known or not. Finally, for the updateGrid function to keep track of when a grid is newly visited this function stores the previous grid values, which are reset to current positions at the end of the callback function(after updateGrid() happens).

The remaining functions in the code are either supplementary functions for the rest of the code or control handlers. These functions and their description are listed below:

- **updateGrid**: This function is responsible for updating the robot perception based on grayscale values set to the grid to make visualisation easier. Whenever called this function checks to see if the current grid value is set to unvisited(50), if that is the case the value will be set to visited once(128). If that grid has already been visited, the function checks to see if the grid was revisited(by comparing current position to previous position), or simply the grid the robot has occupied for some time. If the conclusion is that the grid is revisited, the grid value is increased(by 20). After this the robot perception is saved as an image(which is why an opencv matrix was used for the grid and the values of the grid were as described before instead of starting at 1 and incrementing by 1). At this point the robot resets a timer if the grid is a newly visited grid, this timer is the basis of the next step. After this the function also checks to see if the robot has been stuck in the current grid for more than 20 seconds, if so all adjacent grids are set to blocked to get the robot to stop searching for unexplored grids. Finally, this function checks to see if any grid is visited 4 or more times, in which case the default rotation direction will be flipped(since visiting a grid 4 times is an indication of the robot looping).
- **get_grid_direction**: This function takes an angle value and determines which grid the robot is facing. The direction is calculated based on angle value in the range (0-2π), where each of the eight grids adjacent to the current grid is given 45°(with grid direction identifiers from 0 to 7), as seen below:

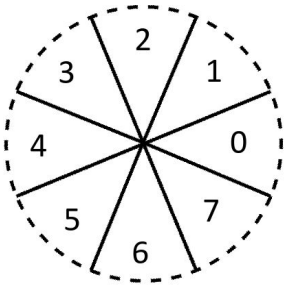

Figure 12: Graphical Representation of The 8 Grid Directions

- **handleBumper**: This function performs exactly as described in the previous sections of the document. If any bumper is pressed every flag is reset and a bumper maneuver takes place(as described in figure 11), and the maneuver identifier is set so that the function can complete the maneuver, also to prevent other functions from performing anything.
- **checkAdj**: This function takes a grid direction(0-7, figure 13), and checks the value of the adjacent grid in the specified direction, and will return a true boolean if the corresponding grid is an unexplored grid, otherwise it returns a false indicating that the corresponding grid is either visited or blocked.

- **assign_blocked**: This function takes a grid direction(0-7, figure 13), and sets the value of the corresponding grid to 0(meaning blocked) if the corresponding grid is unexplored.
- **Log**: This function saves(and updates) the values of the variables of interest in a JSON file, which is then accessed by an HTML page which is used to monitor the robot by the team.
- **handleExloration**: This function is the primary logic function that determines the robots behaviour. Several things are happening in this function.
    - Firstly, the function checks to see if a bumper maneuver is in progress, and if that is the case it will not perform anything.
    - If it is possible to explore and the bumper maneuver is not in progress, then the logic as described in the controller section will take place. Given the code is implemented in a simple way the behaviour described in the controller section is sufficient in properly describing the functions behaviour.
    - Another thing that is done in this function is that if a search is in progress, the function will check to see if any grid directions were omitted during the search, which implies that the grid corresponding to that direction is blocked. In that case the corresponding grid is set to blocked using the assign_blocked function. This is done by checking what grid direction was the direction that was omitted before the current direction and if that direction is not the initial search direction it means that the previous grid was blocked and therefore it must also be set to blocked in the grid values.
- **handleScan**: This function is in charge of scanning and performs exactly as described in the controller description. The important detail here is that during a scan the perception is updated with regards to blocks. This is done in a similar way to what happens in the handleExploration function. However simply passing a grid direction is not indicative of that grid being blocked since a scan will omit all direction, therefore a set of 8 flags(in an array which are reset at the start of each scan) are used during scan to check if any directions are blocked and if that is the case, the corresponding grids will be set to blocked.
- **handle_wall_avoidance**: This function is the part of the code that implements the previously described wall avoidance, where it will check to see if the robot state is one that is not during a bumper maneuver, and based on the left and right minimum distance, it will add corrective angular velocities to avoid wall collisions from the sides(refer to controller description for complete details).

Now that all the functions are described the main code is explained to complete the full picture and provide an overview of the code.

3.2.3.2. The Complete Picture

In this section the main function of the c++ code will be explored and the overview of the entire code and algorithm will be described to paint a full picture of the robot behavior. Inside the main function the following happens in order:

- First ROS is initiated to start working with the robot.
- After ROS is initiated the callbacks are set to subscribe to the appropriate publishers that provide the required sensory data.
- After this the publisher for the velocity settings is initiated and later in the code based on the calculated decisions the appropriate velocities are published.

- After this the refresh rate for the robot is set to 10Hz.
- After this the timer of the code is started both to keep track of time in the decision making process and to keep track of the contest time to prevent overtime and keep the robot running only within 8 minutes.
- After the timer is set the primary loop for robot operations starts, and goes on until contest time is finished.
- Inside this loop first the ros::spinOnce function is called to update the sensor values and trigger the appropriate callbacks, after this the handleScan function is called, which will check to see if a scan is appropriate before scanning, which will prevent conflict. After that the scanning flag is checked to see if a scan is in progress, and in the case of a scan it will skip to the log function, until the scan is finished. If a scan is not in progress the three handler functions will be called, which all have logics that check for conflict and prevent them in the correct priority. These handler functions in order will calculate the final velocities for the robot based on the sensory and perception data available.
- After velocities are calculated the results are published to teleop to have the robot take appropriate actions.
- At this point the variables are saved into the log file, and the timer is updated and the looprate delay is set. After the looprate delay the main loop continues and the process is repeated.

It is noteworthy to mention that a monitoring GUI was also developed using HTML/JS to monitor the robot perception and state in parallel. This GUI is described in more detail in Appendix B.

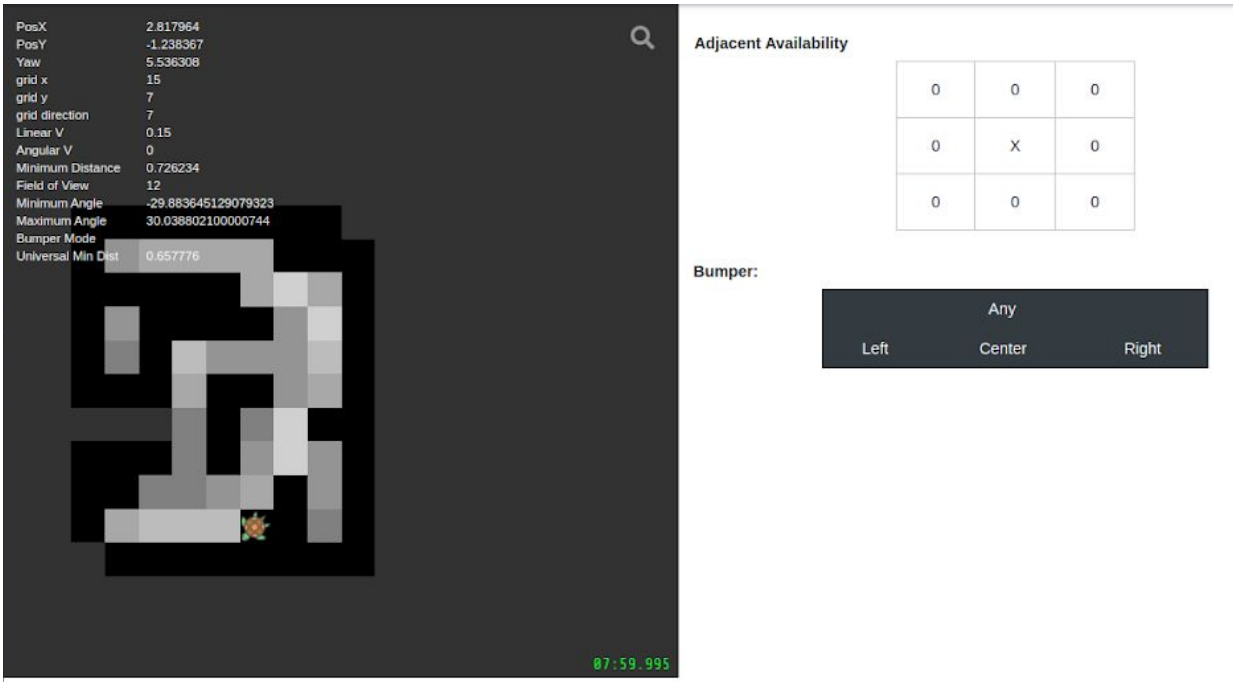

Figure 13: Image of The GUI Environment

Now that the code and algorithms implemented in the code are described, it is important to note the changes that were made to the base skeleton code, as required for this report.

3.2.3.3. Changes Made to The Skeleton Code

GAs evident the changes that were made to the base skeleton code before applying the current code are very limited. These changes are as follows:

- The linear, angular and timer are made into global variables to make them easily accessible to all functions to change speeds and keep track of time for specific maneuvers.
- The timer units were changed from seconds to milliseconds, to be able to measure time more accurately as many of the maneuvers in the code are time based and require more precise timers.

The rest of the changes were in adding new functions and altering callback functions which were described in the prior section.

4. Future Recommendations

In the future, many steps can be taken to improve the searching capabilities of the TurtleBot either through the algorithm developed or hardware installed.

4.1. Wall Following Algorithm

The team had already developed a wall following algorithm that allows the TurtleBot to follow the right walls at all times. However, as mentioned before, a complex feedback system would have to be developed in order to keep the wall to the right of the TurtleBot at all times. Essentially, the TurtleBot would constantly need to readjust itself to be parallel to the wall adding to the overall time, and it was observed that some of the TurtleBots were prone to slight deviations as they moved in a linear fashion. The algorithm was also prone to chaotic behaviour as the TurtleBot approached corners. Furthermore, round objects could not be properly traced in gmapping. However, if given more time, a more sophisticated wall following algorithm could be developed that fixes the issues mentioned. We could have then compared the wall following with the frontier search to see which algorithm maps the maze better. Furthermore, we could have experimented with integrating the frontier search with the wall following to create an amalgamated algorithm that brings together the advantages of both methods, possibly resulting in a very sophisticated searching algorithm.

4.2. TurtleBot Perception

Certain hardware limitations of the TurtleBot have caused limitations with the perception of the TurtleBot with its environment. The odometer present in the TurtleBot is highly unreliable as it accumulates error while running. The errors present in the odometer were significant enough that it could affect the grid system implemented by the team. Since the grid system forms the basis of the traversal decisions made by the algorithm, it was imperative to alleviate this issue. As a result, this became an optimization problem where the team had to balance between having a grid system that was fine enough to properly showcase areas the TurtleBot had not explored and narrow areas, but coarse enough so that the errors from the odometer readings did not affect the traversing and so that the TurtleBot did not spend too much time in a certain area. The team had settled on a grid system of 10 x 10 m which proved to satisfy both of these objectives given the constraint of the error from the odometer. However in an ideal case, the team would have preferred an odometer which had very minimal error in its readings, which would allow us to make the grid a little more refined and potentially give us a more accurate traversal.

4.3. Low-level Controllers and Global Adjustments

During live testing of the TurtleBot, the team noticed that the TurtleBot was severely veering off to the right. This was an issue faced by many of the other groups and simulations run by us in RVIZ showcased the TurtleBot indeed moving in a perfectly linear fashion, therefore there was something possibly faulty with the TurtleBot hardware. This proved to be very frustrating as this veering off can constantly cause the TurtleBot to approach a wall or object. As a result of the TurtleBot approaching an obstacle, this would either cause the TurtleBot to slow down to 0.1 m/s or actually bump against it triggering the bumper mfunction. This adds even more time as the TurtleBot has to constantly slow down or perform the bumper maneuvers, thus preventing us from using the algorithm to its max

efficiency. In order to alleviate this issue in the future and for testing, the TAs suggested some sort of a feedback system would have to be installed onto the TurtleBot in order to alleviate these minor deviations in the linear motion or have a TurtleBot which did not possess such drastic veering.

4.4. Exploration Handler

The exploration system being used looks at only the neighboring grids to make its decisions. However, a more global approach could potentially be implemented where we look at the grids beyond the ones neighbouring it. The only problem is that we would need to properly identify the walls of the maze. However in certain cases, our algorithm was able to properly identify two of the walls, therefore we can immediately constrain the grid by identifying the other two walls. We can then look at the global grid points and make a more informed decision as to which direction the TurtleBot should move so that it directly moves to an unexplored area and also avoids the areas it has visited multiple times. This would significantly increase the searching efficiency, however implementing this sort of complex algorithm could prove to be quite ambitious and potentially unnecessary as 8 minutes is more than enough time for us to map the grid.

5. Attribution Table

Section	Student Names				
	Amin Heyrani Nobari	Farzaam Khorasani-Gerdehkouhi	Naib Gulam	Chandula Dinuka Jaysinghe	Zarak Khan
Problem Definition	ET, FP	ET, FP	ET, FP	RD, MR, ET, FP	ET, FP
Strategy and Motivation	ET, FP	ET, FP	ET, FP	ET, FP	RD, MR, ET, FP
Sensory Design	ET, FP	RS, MR, ET, FP	ET, FP	ET, FP	RS, RD, MR, ET, FP
Controller Design	RS, ET, FP, RD	ET, FP	ET, FP, MR	ET, FP	ET, FP
Future Recommendations	ET, FP	ET, FP	RS, MR, ET, FP	ET, FP	ET, FP
Appendix A: ROS Code	-	-	-	-	All
Appendix B: Robot Monitoring Interface	-	-	-	All	-
Code Development	RS, RD, MR,	RS, ET	RS, ET	RS, ET	RS, ET
Monitoring Interface development	RS, MR	-	RD, MR	-	-

Fill in abbreviations for roles for each of the required content elements. You do not have to fill in every cell. The "all" row refers to the complete report and should indicate who was responsible for the final compilation and final read through of the completed document.

RS – research

RD – wrote first draft

MR – major revision

ET – edited for grammar and spelling

FP – final read through of complete document for flow and consistency

CM – responsible for compiling the elements into the complete document

OR – other

6. References

[1]P. Z. T. Services, “wheel drop sensor: English to German: Automation & Robotics,” *ProZ.com | Freelance translators and interpreters*. [Online]. Available: https://www.proz.com/kudoz/english-to-german/automation-robotics/5100418-wheel-drop-sensor.html. [Accessed: 13-Feb-2020].

[2]C. Woodford, “How do Roomba robot vacuum cleaners work?,” *Explain that Stuff*, 17-Dec-2018. [Online]. Available: https://www.explainthatstuff.com/how-roomba-works.html. [Accessed: 13-Feb-2020].

7. Appendix A: ROS Code

```
#include <ros/console.h>
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <kobuki_msgs/BumperEvent.h>
#include <sensor_msgs/LaserScan.h>
#include <nav_msgs/Odometry.h>
#include <tf/transform_datatypes.h>
#include <stdio.h>
#include <cmath>
#include <chrono>
#include <opencv2/opencv.hpp>
#include <opencv2/highgui.hpp>
#include <fstream>
#include <string.h>
#include <cstdlib>

using namespace cv;
using namespace std;

//Define Macros
#define RAD2DEG(rad) ((rad) * 180. / M_PI)
#define DEG2RAD(deg) ((deg) * M_PI / 180.)
#define DIST(x_1,y_1,x_2,y_2) (sqrt((x_1 - x_2) * (x_1 - x_2) + (y_1 - y_2) * (y_1 - y_2)))

//Set the Constant for The Number of Bumper Switches
#define N_BUMPER (3)
#define Grid_Size 20

//Define Global Variables
float angular = 0.0, linear = 0.0;
float min_dist = 0.7;
float default_lin = 0.2, default_ang = M_PI / 3, scan_ang = M_PI / 6;
float posX = 0.0, posY = 0.0, yaw = 0.0;
int grid_x = 0, grid_y = 0, grid_dir = 0;
uint8_t bumper[3] = {kobuki_msgs::BumperEvent::RELEASED,
kobuki_msgs::BumperEvent::RELEASED, kobuki_msgs::BumperEvent::RELEASED};
float minLaserDist = numeric_limits<float>::infinity();
float LeftLaserDist = numeric_limits<float>::infinity();
float RightLaserDist = numeric_limits<float>::infinity();
float AllminLaserDist = numeric_limits<float>::infinity();
int32_t nLasers = 0, desiredNLasers = 0, desiredAngle = 12.5, desiredNLasersSides = 0,
desiredAngleSides = 12.5;
float max_angle, min_angle;
bool any_bumper_pressed;
int8_t bumper_maneuver = 0;
int temp_g_dir, scanning_g_dir;
uint64_t temp_time;
uint64_t millisecondsElapsed = 0;
uint64_t lastScanTime = 0;
uint64_t lastNewGridTime = 0;
```

```
int prev_x = 0, prev_y = 0;
bool searching = false, all_known = false;
Mat grid(Grid_Size, Grid_Size, CV_8UC1, Scalar(50));
int rotation_direction = 1;
int default_rotation_direction = 1;
bool available_scanned_dirs[8] = {false};
bool scanning = false;
vector<vector<float>> scannedlocations;


//Definition of Supplementary functions
void updateGrid(); //Updates the grid values for exploration at each ros::spin based on
odometry
void handleBumper();//handels bumper events and maneuvers
void handleExloration();//handels the decision making and route selection for exploring the
area
bool checkAdj(int dir);//checks adjacent grid in specific direction(int dir (0-7)) to see
if the neighboring grid in this direction has been visited
void log();//logs the needed variables in a JSON file to be read by the robot monitor UI
int get_grid_direction(float angle);//Calculates the current facing adjacent grid based on
an input angle(0-2PI)
void assign_blocked(int dir);//Assigns the adjacent grid in specific direction(int die
(0-7)) a value of 0 meaning it is blocked
void handle_wall_avoidance();//Handles avoiding side obstacles

//Callback Functions
void bumperCallback(const kobuki_msgs::BumperEvent::ConstPtr& msg)
{//Bumper Operations are based on the settings applied here

   //save bumper states
   bumper[msg->bumper] = msg->state;

   //activate bumper flag if any bumper is pressed
   any_bumper_pressed = false;
   for (uint32_t b_idx = 0; b_idx < N_BUMPER; ++b_idx)
       any_bumper_pressed |= (bumper[b_idx] == kobuki_msgs::BumperEvent::PRESSED);
}

void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{//Most important sensory data is established here
   //set minimum variables to inf for min calculations
   minLaserDist = numeric_limits<float>::infinity();
   AllminLaserDist = numeric_limits<float>::infinity();
   LeftLaserDist = numeric_limits<float>::infinity();
   RightLaserDist = numeric_limits<float>::infinity();

   //get the number of laser sensors and the min and max angles data is available
   nLasers = (msg->angle_max - msg->angle_min) / msg->angle_increment;
   max_angle = msg->angle_max;
   min_angle = msg->angle_min;

   //determine the number of laser sensors that must be read for the desired field of view
   desiredNLasers = desiredAngle * M_PI / (180 * msg->angle_increment);
   desiredNLasersSides = desiredAngleSides * M_PI / (180 * msg->angle_increment);
```

```
   //See if the desired field of view is possible or not and find the minimum distance in
the field of view
   if (desiredAngle * M_PI / 180 < msg->angle_max && -desiredAngle * M_PI / 180 >
msg->angle_min) {
       for (uint32_t laser_idx = nLasers / 2 - desiredNLasers; laser_idx < nLasers / 2 +
desiredNLasers; ++laser_idx){
           minLaserDist = min(minLaserDist, msg->ranges[laser_idx]);
       }
   }
   else {
       for (uint32_t laser_idx = 0; laser_idx < nLasers; ++laser_idx) {
           minLaserDist = min(minLaserDist, msg->ranges[laser_idx]);
       }
   }
   // Determine the minimum distance in all available directions
   for (uint32_t laser_idx = 0; laser_idx < nLasers; ++laser_idx){
           AllminLaserDist = min(AllminLaserDist, msg->ranges[laser_idx]);
   }

   if(desiredNLasersSides < nLasers){//see if desired field of view is possible
       //determine the minimum distance on the sides
       for (uint32_t laser_idx = 0; laser_idx < desiredNLasersSides; ++laser_idx){
           RightLaserDist = min(RightLaserDist, msg->ranges[laser_idx]);
       }
       for (uint32_t laser_idx = nLasers - desiredNLasersSides - 1; laser_idx < nLasers;
++laser_idx){
               LeftLaserDist = min(LeftLaserDist, msg->ranges[laser_idx]);
       }
   }
   else{//if not all lasers must be examined so it would be the same as AllminLaserDist
       LeftLaserDist = AllminLaserDist;
       RightLaserDist = AllminLaserDist;
   }

   //check to ensure laser distance readings are done appropriately (sometimes laser
distances are not available(very close distance or very long ones))
   if(!isfinite(minLaserDist) || minLaserDist < 0)
       minLaserDist = 0;

   if(!isfinite(RightLaserDist) || RightLaserDist < 0)
       RightLaserDist = 0;

   if(!isfinite(LeftLaserDist) || LeftLaserDist < 0)
       LeftLaserDist = 0;

   if(!isfinite(AllminLaserDist) || AllminLaserDist < 0)
       AllminLaserDist = 0;

   //Global velocity determination based on distance measurements
   if(AllminLaserDist <= 0.8 && AllminLaserDist > 0.55){
       default_lin = 0.15;
   }
   else if(AllminLaserDist <= 0.55){
```

```
        default_lin = 0.1;
    }
    else{
        default_lin = 0.2;
    }
    if(bumper_maneuver != 0)//Set to 0.15 m/s for accurate bumper maneuver
        default_lin = 0.15;
}

void odomCallback (const nav_msgs::Odometry::ConstPtr& msg)
{//odometry data is analyzed and on this basis the preception is updated
    //determine the position and based on that determine the grid(setup as 10mx10m grid with
a specified size)
    posX = msg->pose.pose.position.x;
    posY = msg->pose.pose.position.y;
    grid_x = (Grid_Size - 1) / 2 + round(posX / (10.0 / Grid_Size));
    grid_y = (Grid_Size - 1) / 2 + round(posY / (10.0 / Grid_Size));

    //determine yaw
    yaw = tf::getYaw(msg->pose.pose.orientation);
    if(yaw < 0)// since default yaw is in (-PI,PI) change the value to get a yaw in a
(0-2PI) range
        yaw += 2 * M_PI;

    //determine the adjacent grid the robot is facing based on yaw
    grid_dir = get_grid_direction(yaw);

    //update the preception based on the newly attained data
    updateGrid();

    //check to see if all nearest neighbours are known based on the grid data and activate
the appropriate flag if so
    all_known = false;
    for(int i=0;i<8;i++){
        all_known |= checkAdj(i);
    }
    all_known = !all_known;

    //once grid has been updated record update current readings as the previous grid values
so that in the next run it can checked to see if a new grid is visited
    prev_y = grid_y;
    prev_x = grid_x;
}

//Supplementary Functions
void updateGrid(){//Updates the preception(grid system)

    if(grid.at<uchar>(grid_x,grid_y) == 50)//check to see if the current grid is unvisited
if so make it a visited grid(50 is default value of not visited The value was chosen for
the picture to look readable)
        grid.at<uchar>(grid_x,grid_y) = 128;//default value of visited once is 128(for
readability of the image)
```

```
   else if(grid.at<uchar>(grid_x,grid_y) >= 128 && (prev_x != grid_x || prev_y != grid_y)
&& grid.at<uchar>(grid_x,grid_y) <= 235){//check to see the current grid is newly visited
and is not simply the same as before
       grid.at<uchar>(grid_x,grid_y) += 20;//add 20 to the pixel value of the grid(20 based
on readability)
   }
   if (prev_x != grid_x || prev_y != grid_y)//update the time of visit to the new grid is a
new grid is visited
       lastNewGridTime = millisecondsElapsed;

   imwrite("current.png", grid);//Save an image of the robots current preception for
monitoring

   if(millisecondsElapsed - lastNewGridTime >= 30000){//check to see if the robot is stuck
at specific grid location and assign blocks to all surrounding unknown grids(which is
likely why the robot is stuck anyways)
       for(int i = 0;i<8;i++)
           assign_blocked(i);
   }

   if(grid.at<uchar>(grid_x,grid_y) >= 188){//If at any point a grid location is visited
for the fourth time(implies the robot is stuck) the default rotation direction of the robot
is flipped
       default_rotation_direction = -1;
   }
}

int get_grid_direction(float angle){
   //find the corresponding grid direction for the current yaw value
   if((angle >= -M_1_PI/8 && angle <= M_PI/8) || (angle >= 2 * M_PI - M_PI/8 && angle <= 2
* M_PI + M_PI/8))
       return 0;
   if((angle >= M_PI/8 && angle<= 3 * M_PI/8) || (angle >= 2 * M_PI + M_PI/8))
       return 1;
   if(angle >= 3 * M_PI/8 && angle<= 5 * M_PI/8)
       return 2;
   if(angle >= 5 * M_PI/8 && angle<= 7 * M_PI/8)
       return 3;
   if(angle >= 7 * M_PI/8 && angle<= 9 * M_PI/8)
       return 4;
   if(angle >= 9 * M_PI/8 && angle<= 11 * M_PI/8)
       return 5;
   if(angle >= 11 * M_PI/8 && angle<= 13 * M_PI/8)
       return 6;
   if((angle >= 13 * M_PI/8 && angle<= 15 * M_PI/8) || (angle <= -M_PI/8))
       return 7;
}

void handleBumper(){
   //First check if bumper is pressed or another maneuver in progress
   if(any_bumper_pressed && bumper_maneuver == 0){
       //Reset Everything and decide which bumper was pressed and what maneuver is
appropriate
       searching =false;
```

```
        temp_time = millisecondsElapsed;
        if(bumper[kobuki_msgs::BumperEvent::CENTER] == kobuki_msgs::BumperEvent::PRESSED){
            angular = 0.0;
            linear = -default_lin;
            bumper_maneuver = 1;
        }
        if(bumper[kobuki_msgs::BumperEvent::RIGHT] == kobuki_msgs::BumperEvent::PRESSED){
            linear = -default_lin;
            angular = 0.0;
            bumper_maneuver = 3;
        }
        if(bumper[kobuki_msgs::BumperEvent::LEFT] == kobuki_msgs::BumperEvent::PRESSED){
            linear = -default_lin;
            angular = 0.0;
            bumper_maneuver = 2;
        }
    }

    //check to see if another maneuver is in progress and act accordingly to complete the
maneuver(described in the report)
    else if(bumper_maneuver == 1 && millisecondsElapsed - temp_time >= 2000){
        bumper_maneuver = 0;
        linear = 0.0;
        angular = 0.0;
    }
    else if(bumper_maneuver == 2 && millisecondsElapsed - temp_time >= 1500){
        angular = -default_ang;
        if(millisecondsElapsed - temp_time >= 2000){
            linear = default_lin;
            angular = 0.0;
            if(millisecondsElapsed - temp_time >= 3000){
                bumper_maneuver = 0;
                linear = 0.0;
                angular = 0.0;
            }
        }
    }
    else if(bumper_maneuver == 3 && millisecondsElapsed - temp_time >= 1500){
        linear = linear / 2;
        angular = default_ang;
        if(millisecondsElapsed - temp_time >= 2000){
            linear = default_lin;
            angular = 0.0;
            if(millisecondsElapsed - temp_time >= 3000){
                bumper_maneuver = 0;
                linear = 0.0;
                angular = 0.0;
            }
        }
    }
}

bool checkAdj(int dir){
    //first determine the increments of the corresponding grid direction
```

```
   int x_f = 0, y_f = 0;
   if(dir == 0 || dir == 1 || dir == 7)
       x_f = 1;
   else if(dir == 3 || dir == 4 || dir == 5)
       x_f = -1;
   else
       x_f = 0;

   if(dir == 1 || dir == 2 || dir == 3)
       y_f = 1;
   else if(dir == 5 || dir == 6 || dir == 7)
       y_f = -1;
   else
       y_f = 0;

   if(grid_x + x_f >= 0 && grid_x + x_f < Grid_Size && grid_y + y_f >= 0 && grid_y + y_f <
Grid_Size){//check to see if desired grid is within the grid system or past it's boundaries
       if(grid.at<uchar>(grid_x + x_f, grid_y + y_f) == 50){//if the grid is unvisited
return a true value
           return true;
       }
   }
   return false; // in any other case return a false value indicating either visited or
unavailable
}

void handleExloration(){
   if(searching){//if searching check if any adjacent grids where blocked and if so change
the values
       //similar to scanning logic for block detection refer to handleScan function
       if(grid_dir != temp_g_dir){
           int prev_grid_dir = grid_dir - rotation_direction;
           if(prev_grid_dir < 0)
               prev_grid_dir = 7;
           if(prev_grid_dir > 7)
               prev_grid_dir = 0;
           if(prev_grid_dir != temp_g_dir)
               assign_blocked(prev_grid_dir);
       }
   }
   if(bumper_maneuver == 0){// check to see if any bumper operation in progress
       if(minLaserDist >= min_dist && checkAdj(grid_dir)){// if not blocked and facing an
unexplored grid continue motion or stop searching
           angular = 0.0;
           linear = default_lin;
           searching =false;
       }
       else if(minLaserDist >= min_dist && ((millisecondsElapsed - temp_time >= 6000 &&
searching) || all_known || grid.at<uchar>(grid_x,grid_y) >= 168)){
           //check to see if forward direction available and either full 360 search has
been completed or all adjacent grids are know or the current grid has been visited 3 or
more times
           //if so move forward and don't waste time searching more
           angular = 0.0;
```

```
            linear = default_lin;
            searching = false;
        }
        else{//if the forward direction is blocked
            if(!searching){//see if a search is in progress and only go into search if not
already searching
                if(!all_known){//if all neighbouring grids are not known
                    searching = true;//start search
                    temp_time = millisecondsElapsed;//reset 360 timers
                    temp_g_dir = grid_dir;//save the starting grid(for block checking)

                    //see if only one unexplored neighbour exists
                    int nAvailable = 0, available_dir;
                    for(int i=0;i<8;i++){
                        if(checkAdj(i)){
                            nAvailable ++;
                            available_dir = i;
                        }
                    }

                    if(nAvailable == 1){//if only one unexplored neighbour exists then find
the fastest direction of rotation
                        //calculate which rotation direction would be faster
                        int forward,backward;
                        forward = available_dir - grid_dir;
                        backward = grid_dir - available_dir;
                        if(forward < 0)
                            forward += 8;
                        if(backward < 0)
                            backward += 8;
                        if(backward<forward)
                            rotation_direction = -default_rotation_direction;
                        else if(available_dir == grid_dir){
                            //if the currently facing direction is the only available and
blocked(because we are here) set the facing grid to blocked
                            assign_blocked(grid_dir);

                            //rotate in the default direction
                            rotation_direction = default_rotation_direction;
                        }
                        else
                            rotation_direction = default_rotation_direction;//in any other
case rotate in the default direction

                    }
                    else{// in other cases rotate in default direction
                        rotation_direction = default_rotation_direction;
                    }
                }
            }
            if(all_known)
                rotation_direction = default_rotation_direction;//if all known rotate in
default direction(this is here to reset rotation direction if one was known and it has been
established that it was blocked)
```

```
            //at the end just set the velocities based on the logit prior
            angular = rotation_direction * default_ang;
            linear = 0.0;
        }
    }

}

void assign_blocked(int dir){//assign the neighbour grid in the corresponding grid
direction as blocked
    //determine increments based on specified direction
    int x_f = 0, y_f = 0;
    if(dir == 0 || dir == 1 || dir == 7)
        x_f = 1;
    else if(dir == 3 || dir == 4 || dir == 5)
        x_f = -1;
    else
        x_f = 0;

    if(dir == 1 || dir == 2 || dir == 3)
        y_f = 1;
    else if(dir == 5 || dir == 6 || dir == 7)
        y_f = -1;
    else
        y_f = 0;


    if(grid_x + x_f >= 0 && grid_x + x_f < Grid_Size && grid_y + y_f >= 0 && grid_y + y_f <
Grid_Size){//check to see if in bound
        if(grid.at<uchar>(grid_x + x_f, grid_y + y_f) == 50){//if grid is unvisited set the
grid to blocked(0)
            grid.at<uchar>(grid_x + x_f, grid_y + y_f) = 0;
        }
    }
}

void log(){
    //Saves a JSON file with the relevant information later to be monitored in an HTML log
page
    String temp;
    ofstream logfile("log.json");
    if (logfile.is_open())
    {
        logfile << "{";
        temp ="\"angular\": " + to_string(angular) + ",";
        logfile << temp;
        temp ="\"linear\": " + to_string(linear) + ",";
        logfile << temp;
        temp ="\"posX\": " + to_string(posX) + ",";
        logfile << temp;
        temp ="\"posY\": " + to_string(posY) + ",";
        logfile << temp;
        temp ="\"yaw\": " + to_string(yaw) + ",";
```

```
       logfile << temp;
       temp ="\"grid_x\": " + to_string(grid_x) + ",";
       logfile << temp;
       temp ="\"grid_y\": " + to_string(grid_y) + ",";
       logfile << temp;
       temp ="\"grid_dir\": " + to_string(grid_dir) + ",";
       logfile << temp;
       temp ="\"minLaserDist\": " + to_string(minLaserDist) + ",";
       logfile << temp;
       temp ="\"desiredAngle\": " + to_string(desiredAngle) + ",";
       logfile << temp;
       temp ="\"any_bumper_pressed\": " + to_string(any_bumper_pressed) + ",";
       logfile << temp;
       temp ="\"millisecondsElapsed\": " + to_string(millisecondsElapsed) + ",";
       logfile << temp;
       temp ="\"searching\": " + to_string(searching) + ",";
       logfile << temp;
       temp ="\"bumper_maneuver\": " + to_string(bumper_maneuver) + ",";
       logfile << temp;
       temp ="\"Grid_size\": " + to_string(Grid_Size) + ",";
       logfile << temp;
       temp ="\"min_angle\": " + to_string(min_angle) + ",";
       logfile << temp;
       temp ="\"max_angle\": " + to_string(max_angle) + ",";
       logfile << temp;
       temp ="\"all_min\": " + to_string(AllminLaserDist) + ",";
       logfile << temp;
       temp ="\"scores\": [" + to_string(checkAdj(0))  + "," + to_string(checkAdj(1))  +
"," + to_string(checkAdj(2))  + "," + to_string(checkAdj(3))  + "," +
to_string(checkAdj(4))  + "," + to_string(checkAdj(5))  + "," + to_string(checkAdj(6))  +
"," + to_string(checkAdj(7)) + "],";
       logfile << temp;
       temp ="\"bumpers\": [" + to_string(bumper[kobuki_msgs::BumperEvent::RIGHT] ==
kobuki_msgs::BumperEvent::PRESSED) + "," +
to_string(bumper[kobuki_msgs::BumperEvent::CENTER] == kobuki_msgs::BumperEvent::PRESSED) +
"," + to_string(bumper[kobuki_msgs::BumperEvent::LEFT] ==
kobuki_msgs::BumperEvent::PRESSED) + "]}";
       logfile << temp;
       logfile.close();
   }
}

void handleScan()
{//Scanning is done here
   //set a temp value to infinity to check the minimum distance to any scan locations
   float temp_distance = numeric_limits<float>::infinity();

   //find minimum distance to any scan locations
   for(int i = 0; i<scannedlocations.size();i++){
       if(lastScanTime != 0)
           temp_distance = min(DIST(posX,posY,scannedlocations[i][0],
scannedlocations[i][1]), temp_distance);
   }
```

```
   if(!searching && bumper_maneuver == 0 && ((temp_distance >= 1.8 &&
isfinite(temp_distance)) || millisecondsElapsed - lastScanTime >= 150000) &&
!scanning){//check to see if scanning can be done at this moment
       //scanning if more 2.5 minutes from last scan or distance more than 1.8m from any
scan location
       lastScanTime = millisecondsElapsed; // reset the scan timer
       vector<float> temp_pos_vec = {posX,posY}; //make a float vector of the current
position
       scannedlocations.push_back(temp_pos_vec);//save the scan location to be checked
later
       scanning_g_dir = grid_dir;

       //set the scan velocities
       linear = 0.0;
       angular = scan_ang;

       //activate scanning flag
       scanning = true;

       //reset availability flags for block checking
       for(int i=0;i<8;i++)
           available_scanned_dirs[i] = false;
   }
   else if(scanning &&  millisecondsElapsed - lastScanTime >= 2000 * M_PI /
scan_ang){//check to see if scan has been completed
       scanning = false;
       angular = 0.0;
       linear = 0.0;
   }
   else if(scanning){//while scanning see if any adjacent blocks can be detected
       if(minLaserDist >= min_dist)//if not blocked activate the availability flag
           available_scanned_dirs[grid_dir] = true;
       if(grid_dir != scanning_g_dir){//check to see if an adjacent blocked a=has been
fully scanned
           int prev_grid_dir = grid_dir - 1; //calculate the previous

           //the following is necessary correction to find prev direction in the
appropriate (0-7) range
           if(prev_grid_dir < 0)
               prev_grid_dir = 7;
           if(prev_grid_dir > 7)
               prev_grid_dir = 0;

           //if an adjacent grid is scanned and not available set it to blocked
           if(prev_grid_dir != scanning_g_dir && !available_scanned_dirs[prev_grid_dir])
               assign_blocked(prev_grid_dir);
       }
   }
}

void handle_wall_avoidance(){
   //here the wall avoidance augmentation is applied
   if(bumper_maneuver == 0 && linear  > 0){//check to see if not bumper maneuver in
progress and motion in linear exists
```

```
        if(LeftLaserDist < min_dist * 1.7 && RightLaserDist < min_dist * 1.7){//check to see
if in a tight spot
            angular = 0;
        }
        else if(RightLaserDist < min_dist * 1.5){//check to see if too close to the right
            angular = M_PI / 12;//apply correction to avoid obstacles
        }
        else if(LeftLaserDist < min_dist * 1.5){//check to see if too close to the left
            angular = -M_PI / 12;//apply correction to avoid obstacles
        }
        else{//in any other case set to default
            angular = 0;
        }
    }
}


//Main Function
int main(int argc, char **argv)
{
    //ROS init
    ros::init(argc, argv, "maze_explorer");
    ros::NodeHandle nh;
    //Establish Subscribers
    ros::Subscriber bumper_sub = nh.subscribe("mobile_base/events/bumper", 10,
&bumperCallback);
    ros::Subscriber laser_sub = nh.subscribe("scan", 10, &laserCallback);
    ros::Subscriber odom = nh.subscribe("odom", 1, &odomCallback);

    //Start Publishers
    ros::Publisher vel_pub = nh.advertise<geometry_msgs::Twist>("cmd_vel_mux/input/teleop",
1);

    //Set Refresh Rate
    ros::Rate loop_rate(10);

    //Define Vel Object
    geometry_msgs::Twist vel;

    // contest count down timer
    chrono::time_point<chrono::system_clock> start;
    start = chrono::system_clock::now();
    while(ros::ok() && millisecondsElapsed <= 480000) {
        ros::spinOnce();
        //Primary Logic And Operations//
        //handle scan first(see if scanning is possible and appropriate)
        handleScan();
        if(!scanning){//check to ensure scanning not in progress
            //handle bumper operations
            handleBumper();
            //make decisions based on sensor and preception
            handleExloration();
            //apply the final augmentation to the decision based on sensory data
            handle_wall_avoidance();
```

```
        }
        //Setting Velocities
        vel.angular.z = angular;
        vel.linear.x = linear;
        vel_pub.publish(vel);
        //save a log of current variables
        log();
        // The last thing to do is to update the timer.
        millisecondsElapsed =
chrono::duration_cast<chrono::milliseconds>(chrono::system_clock::now()-start).count();
        loop_rate.sleep();
    }

    return 0;
}
```

Appendix B: Monitoring and Debugging Interface

As mentioned in the body of the report, an HTML UI was developed that would show the robot perception, basically the occupancy grid, and display the robot parametres and the states of the adjacent grids, by looking at the image of the grid and the JSON file that is saved by the log function. The Interface looks similar to picture displayed below:

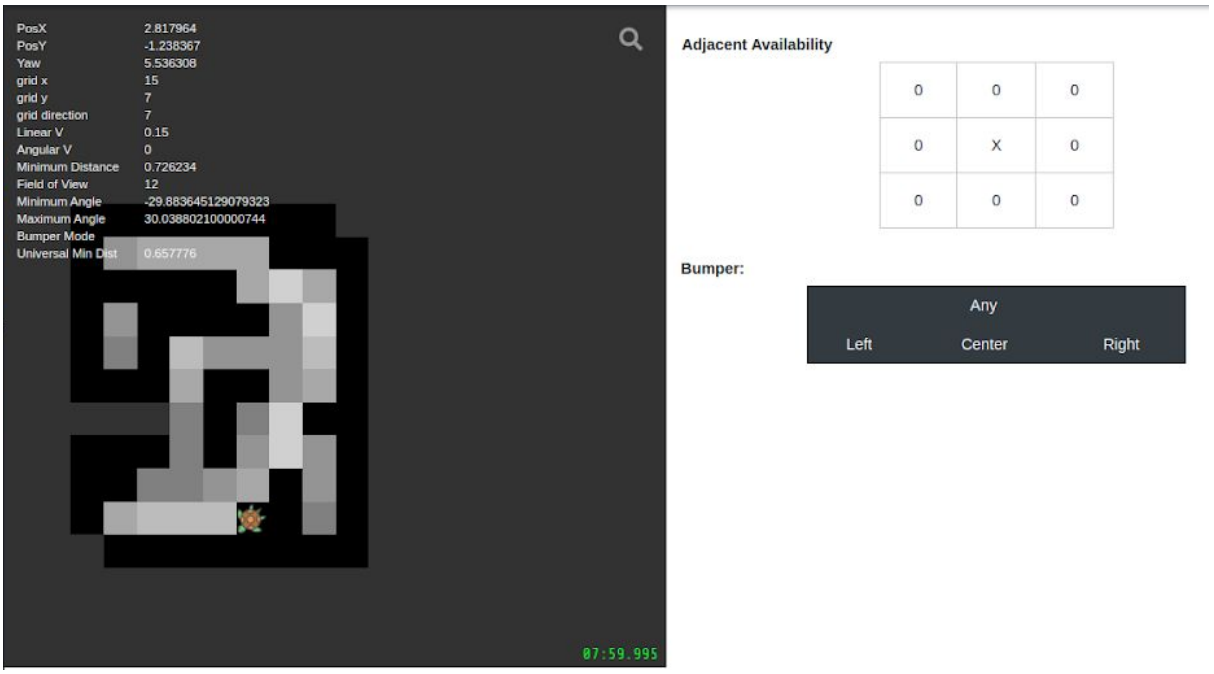

Figure B1: Entire Interface

This interface has two part the left part which include the occupancy grid and the robot parameters and a timer:

Figure B2: Right Side of The Interface

The right side of the interface shows the state of the bumper and the whether the adjacent grids are unexplored:

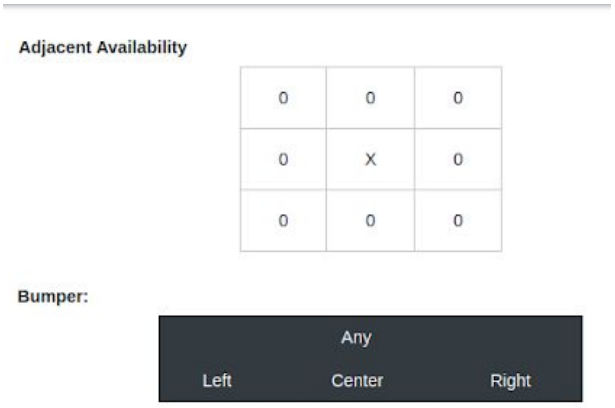

Figure B3: Left Interface

The code for the html and javascript is as follows:

```
<html lang="en">
  <head>
        <!-- Required meta tags -->
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
        <!-- Bootstrap CSS -->
        <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2
MZw1T" crossorigin="anonymous">
        <link
href="https://fonts.googleapis.com/css?family=Share+Tech+Mono&display=swap"
rel="stylesheet">
        <title>   Turlebot Monitor</title>
  </head>
  <body>
        <div class="row no-gutters">
                <div class="col-6 map">
                        <img id="map" src="current.png">
                        <div class="bot">
                                <img src="bot.png">
                        </div>
                        <div class="searching">
                                <i class="fas fa-search"></i>
                        </div>
                        <div class="timer">
                                00:00
                        </div>
                        <div class="pos-inf">
                                <table>
                                        <tr>
                                                <td>PosX</td>
                                                <td id="posx">0.00</td>
                                        </tr>
                                        <tr>
                                                <td>PosY</td>
                                                <td id="posy">0.00</td>
                                        </tr>
                                        <tr>
                                                <td>Yaw</td>
                                                <td id="yaw">0.00</td>
                                        </tr>
                                        <tr>
                                                <td>grid x</td>
                                                <td id="gx">0.00</td>
```

```
                    </tr>
                        <td>grid y</td>
                        <td id="gy">0.00</td>
                    </tr>
                    <tr>
                        <td>grid direction</td>
                        <td id="gd">0.00</td>
                    </tr>
                    <tr>
                        <td>Linear V</td>
                        <td id="lv">0.00</td>
                    </tr>
                    <tr>
                        <td>Angular V</td>
                        <td id="av">0.00</td>
                    </tr>
                    <tr>
                        <td>Minimum Distance</td>
                        <td id="md">0.00</td>
                    </tr>
                    <tr>
                        <td>Field of View</td>
                        <td id="fov">0.00</td>
                    </tr>
                    <tr>
                        <td>Minimum Angle</td>
                        <td id="mina">0.00</td>
                    </tr>
                    <tr>
                        <td>Maximum Angle</td>
                        <td id="maxa">0.00</td>
                    </tr>
                    <tr>
                        <td>Bumper Mode</td>
                        <td id="bm">0.00</td>
                    </tr>
                    <tr>
                        <td>Universal Min Dist</td>
                        <td id="umd">0.00</td>
                    </tr>
                </table>
        </div>
</div>
<div class="col-6 log">
        <div class="col-12 scores">
            <h3>Scores:</h3>
            <table>
                <tr>
```

```
                                        <td id="s5">3</td>
                                        <td id="s4">2</td>
                                        <td id="s3">1</td>
                                </tr>
                                <tr>
                                        <td id="s6">4</td id="s4">
                                        <td>X</td>
                                        <td id="s2">0</td>
                                </tr>
                                <tr>
                                        <td id="s7">5</td>
                                        <td id="s0">6</td>
                                        <td id="s1">7</td >
                                </tr>
                        </table>
                </div>
                <div class="col-12 bumper">
                        <h3>Bumper:</h3>
                        <table class="table table-dark">
                                <tr>
                                        <td class="" id="ba-1"></td>
                                        <td class="" id="ba">Any</td>
                                        <td class="" id="ba-2"></td>
                                </tr>
                                <tr>
                                        <td class="" id="bl">Left</td>
                                        <td class="" id="bc">Center</td>
                                        <td class="" id="br">Right</td>
                                </tr>
                        </table>
                </div>
        </div>
        <script src="https://code.jquery.com/jquery-3.4.1.min.js"
integrity="sha256-CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFlBw8HfCJo="
crossorigin="anonymous"></script>
        <script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIH
NDz0W1" crossorigin="anonymous"></script>
        <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM
" crossorigin="anonymous"></script>
        <script src="https://kit.fontawesome.com/54847ad262.js"
crossorigin="anonymous"></script>
        <script>
                var imageelement = document.getElementById("map");
                var src = "current.png";
```

```
		var log_dump;
		function init() {
			imageelement.src = src + "?t=" + (Math.random() * Math.random() *
100).toString();
		}
		setInterval(init,500);

		function load_log(){
			$.ajax({
				type: "POST",
				url: "log.json",
				contentType:"application/json",
				crossDomain : true,
				success: function(response, textStatus, jqXHR) {
					log_dump = JSON.parse(response);
					render();
				}
			});
		}

		function render(){
			$("#posx").empty();
			$("#posy").empty();
			$("#yaw").empty();
			$("#gx").empty();
			$("#gy").empty();
			$("#gd").empty();
			$("#md").empty();
			$("#lv").empty();
			$("#av").empty();
			$("#fov").empty();
			$("#s0").empty();
			$("#s1").empty();
			$("#s2").empty();
			$("#s3").empty();
			$("#s4").empty();
			$("#s5").empty();
			$("#s6").empty();
			$("#s7").empty();
			$("#bm").empty();
			$("#mina").empty();
			$("#maxa").empty();
			$("#umd").empty();
			$("#posx").append(log_dump["posX"]);
			$("#posy").append(log_dump["posY"]);
			$("#yaw").append(log_dump["yaw"]);
			$("#gx").append(log_dump["grid_x"]);
			$("#gy").append(log_dump["grid_y"]);
```

```
                        $("#gd").append(log_dump["grid_dir"]);
                        $("#md").append(log_dump["minLaserDist"]);
                        $("#lv").append(log_dump["linear"]);
                        $("#av").append(log_dump["angular"]);
                        $("#fov").append(log_dump["desiredAngle"]);
                        $("#s0").append(log_dump["scores"][0]);
                        $("#s1").append(log_dump["scores"][1]);
                        $("#s2").append(log_dump["scores"][2]);
                        $("#s3").append(log_dump["scores"][3]);
                        $("#s4").append(log_dump["scores"][4]);
                        $("#s5").append(log_dump["scores"][5]);
                        $("#s6").append(log_dump["scores"][6]);
                        $("#s7").append(log_dump["scores"][7]);
                        $("#bm").append(log_dump["bumper_manouver"]);
                        $("#umd").append(log_dump["all_min"]);
                        $("#mina").append(log_dump["min_angle"] * 180 / Math.PI);
                        $("#maxa").append(log_dump["max_angle"] * 180 / Math.PI);
                        $(".timer").empty();
                        $(".timer").append('0' +
(Math.floor(log_dump["millisecondsElapsed"]/60000)).toString() + ":" +
((log_dump["millisecondsElapsed"]%60000)/1000).toString());
                        $(".bot").css({
                                "top" : (log_dump["grid_x"] / log_dump["Grid_size"] *
100).toString() + "%",
                                "left" : (log_dump["grid_y"] / log_dump["Grid_size"] *
100).toString() + "%",
                                "height" : (100 / log_dump["Grid_size"]).toString() + "%",
                                "width" : (100 / log_dump["Grid_size"]).toString() + "%",
                                "transform" : "rotate(" + (-log_dump["yaw"] * 360 /
6.28318530718).toString() + "deg)"
                        });

                        if(log_dump["searching"] === 1)
                                $(".searching").addClass("active");
                        else
                                $(".searching").removeClass("active");

                        if(log_dump["any_bumper_pressed"]){
                                $("#ba-1").addClass("bg-warning");
                                $("#ba").addClass("bg-warning");
                                $("#ba-2").addClass("bg-warning");
                        }
                        else{
                                $("#ba-1").removeClass("bg-warning");
                                $("#ba").removeClass("bg-warning");
                                $("#ba-2").removeClass("bg-warning");
                        }
```

```
                if(log_dump["bumpers"][0])
                        $("#br").addClass("bg-warning");
                else
                        $("#br").removeClass("bg-warning");

                if(log_dump["bumpers"][1])
                        $("#bc").addClass("bg-warning");
                else
                        $("#bc").removeClass("bg-warning");

                if(log_dump["bumpers"][2])
                        $("#bl").addClass("bg-warning");
                else
                        $("#bl").removeClass("bg-warning");
        }
        setInterval(load_log,10);
</script>
<style>
        #map{
                image-rendering: optimizeSpeed;
                image-rendering: -moz-crisp-edges;
                image-rendering: -o-crisp-edges;
                image-rendering: -webkit-optimize-contrast;
                image-rendering: pixelated;
                image-rendering: optimize-contrast;
                -ms-interpolation-mode: nearest-neighbor;
                width: 100%;
        }
        #map_block{
                image-rendering: optimizeSpeed;
                image-rendering: -moz-crisp-edges;
                image-rendering: -o-crisp-edges;
                image-rendering: -webkit-optimize-contrast;
                image-rendering: pixelated;
                image-rendering: optimize-contrast;
                -ms-interpolation-mode: nearest-neighbor;
                width: 100%;
                position: absolute;
                top: 0;
                left: 0;
                opacity: 0.5;
        }
        .map{
                position: relative;
        }
        .pos-inf {
                position: absolute;
                top: 0;
```

```
        color: #fff;
}

.pos-inf td {
        padding: 0.2em 1em;
}

.pos-inf tr {
        color: #fff;
        font-size: 0.8em;
        /* padding: 1em; */
}

.pos-inf table {
        margin: 1em 0 0;
}
h3 {
        font-size: 16px;
        font-weight: bold;
}

.scores table {
        text-align: center;
        margin: auto;
}

.scores table td {
        width: 6%;
        line-height: 3.5;
        border: 1px solid #c5c5c5;
}

.col-12.scores {
        padding: 2em 1em;
}
.bumper table {
        text-align: center;
        width: 60%;
        margin: auto;
}
.searching {
        position: absolute;
        top: 1em;
        right: 1em;
        font-size: 1.5em;
        color: #fff;
        opacity: 0.5;
}
```

```
                .d-searching {
                        position: absolute;
                        top: 1em;
                        right: 2.5em;
                        font-size: 1.5em;
                        color: #fff;
                        opacity: 0.5;
                }

                .d-searching.active,
                .searching.active{
                        opacity:1;
                }

                .bot {
                        height: 1.96%;
                        width: 1.96%;
                        position: absolute;
                        top: 47%;
                        left: 47%;
                        transition: all ease 0.2s;
                        text-align: center;
                }

                .bot img {
                        width: 100%;
                        display: inline;
                        margin-top: 10%;
                }

                .timer {
                        position: absolute;
                        bottom: 0.1em;
                        font-family: 'Share Tech Mono', monospace;
                        right: 0.5em;
                        color: lime;
                }
                table.table.table-dark td {
                        border: none;
                }
        </style>
  </body>
</html>
```