Assignment 3
B39VT Integrated Group Robotics Project

1 INTRODUCTION

Disaster – whether natural or manmade – has unfortunately been a persistent occurrence in the history of planet Earth. Earthquakes and tsunamis such as those that occurred in Haiti and the Indian Ocean have taken upwards of 600,000 lives since the beginning of the millennium alone.

It is this horrendous loss of life that has spurred engineers, scientists and emergency services all over the world to increasingly turn to robotic systems in order to aid with disaster response and recovery.

This assignment aims to simulate a robotic solution to a search and rescue operation during a nuclear disaster. The end result should demonstrate the following: the ability to move autonomously in an unknown environment, create and store a map of the given environment, avoid obstacles in the path of the robot, detect and return the position of 6 “individuals” in the area.

Helmi Fraser
H00152077

2 APPROACH

The approach taken for this task was to formulate a solution from the ground up, working in layers and increasing complexity.

To start with, an overall search method was decided. The robot needs to perform the task as optimally as possible i.e. in the least amount of time, with the least amount of collisions, the correct number of individuals detected and without going out of the operation boundary. To this effect, a "lawnmower" (Figure 1) search pattern was discounted, as this would compound the small but significant errors present in the robot's movement leading to massive errors towards the end of the operation, as well as having a high execution time as the robot would be travelling a relatively large total distance. In the end, the search pattern shown in Figure 2 was settled upon. This provides a desirable mix of error minimization, execution speed and total search area discoverability.

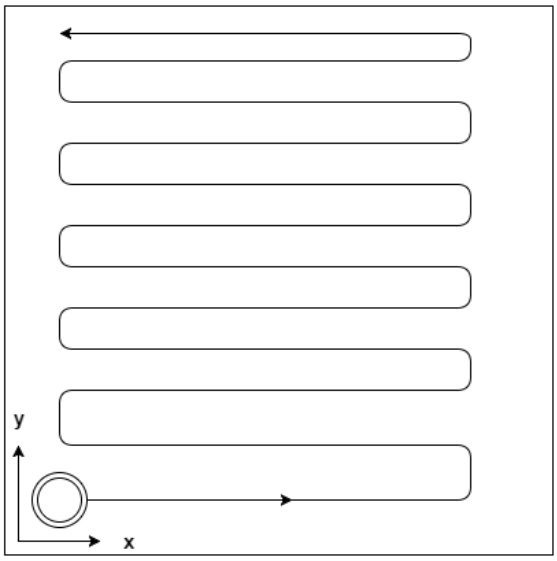


Figure 1 Lawnmower pattern

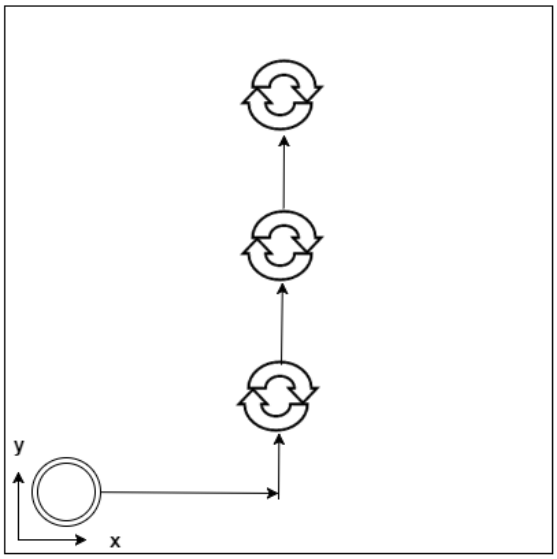


Figure 2 Search pattern

In order to execute the search pattern shown in Figure 2, it is crucial that the foundations of the solution first be developed. This includes creating functions that utilizes the robot's odometry, which allows the robot to execute accurate turns and moving to a set distance. Chaining these functions correctly will form the basis of the robot's movement capabilities.

Figure 3 shows how the angular pose is represented with regard to the odometry values.

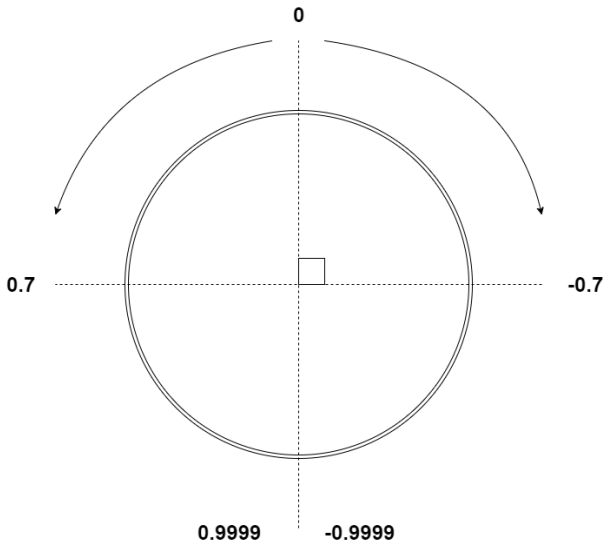


Figure 3 Odometry values

It must be noted that all odometry, be they linear or angular pose is relative from the robot's initial starting location and pose. As can be seen from Figure 3, straight north relative to the starting pose is a value of zero. Turning east from relative north decreases the odometry value from 0 to -1 at relative south. However, it is not consistent as a 90 degree turn in this direction from relative north yields an odometry value of -0.7. This is similar for the other half of the robot, with the exception that the odometry increases from 0 at relative north to 1 at relative south.

It was vital that this was taken into account when the turning functions were developed, as not doing so would lead to miscalculating the target pose and result in wildly inaccurate turns.

It was also vital that any desired angle of turn was converted into the correct odometry values, especially for the 90 degree turns. This was achieved by using trigonometry to calculate the target value with respect to the starting position. A derivation for these equations can be seen in Appendix 1.

Also a basic functionality is the robot's ability to gather visual data from its environment. Colour detection was relatively straight forward, as the majority of the work was completed in the previous assignment and it just needed implementation within the scope of the current assignment. Detecting depth, however, is trickier. As a 3D camera, the robot's Kinect is capable of discerning depth, however this is limited in its range, field of view and wholly dependent on the level of light in the operating area. For this reason, a Hokuyo laser scanner was fitted onto the robot and used for depth detection. This was chosen as the range, field of view and accuracy far outstrips that of the Kinect with the added bonus of being brightness independent, working just as well in complete darkness as it does in light.

With regards to map building and returning objects' locations, the Kinect was used to identify what kind of object is being detected, while data from the Hokuyo was used to calculate the distance to the object. Utilizing trigonometry, this distance was then cross-referenced with the robot's position provided by odometry to calculate x and y co-ordinates of the object detected. By the end of the operation, clusters of co-ordinates will be obtained and these clusters can be averaged which will provide accurate positions of all the objects detected. These co-ordinates, as well as the robot's path and current position can be visually displayed using Rviz.

After developing these, it was then a case of adequately implementing them together at a higher level to produce the desired behavior.

3 DESCRIPTION AND CODE OVERVIEW

The system comprises of lower and higher level functions.

As described in the previous section, the lower level functions are those that by themselves, do relatively simple operations and are pivotal for the solution to function at all. These include: receiving and processing odometry, laser and Kinect data, turning 90 degrees, turning 360 degrees, publishing velocity commands, moving forward and backward linearly for a given distance, outputting data displayable by Rviz and post processing the gathered co-ordinate data.

The odometry data is passed into the program via the `odomcallback()` function. This function takes in an argument of `const nav_msgs::Odometry::ConstPtr& odom_msg` which points to the odometry message. Within the body, the current x and y co-ordinates of the robot and current angle, z, stored in the message are copied to global variables for use in other sections of the program. The function `createline()` plots the path that the robot has taken in Rviz, `createshape()` creates a simple shape inside Rviz to represent the robot's current position, and `search()` is the higher level movement pattern executed by the robot, which will be explained further in this section.

The laser data is passed into the program via the `lasercallback()` function. This function takes in an argument of `const sensor_msgs::LaserScan::ConstPtr& scan` which points to the LaserScan message. Within this function, for loops are used to obtain sensor readings within the region of interest and save them to local arrays i.e. within a certain distance to the robot and within a certain field of view. This data is then analysed and the function performs several checks - based on the value of distances it has detected – and sets several global booleans `(clear_path, clear_turn)` accordingly. All this data is then published to "/base_laser_link" for visualization in Rviz.

The Kinect image data is passed into the program via the `imagecallback()` function, which is largely similar to the code developed for the previous assignment. This function takes in an argument of `const sensor_msgs::ImageConstPtr& msg` which points to the image message. After instantiating several variables and the Mat structures that will hold the hold the original BGR, HSV and processed image values, the function copies the current image seen by the Kinect, in order to make it possible to be manipulated. This image is then converted from BGR to HSV encoding, which is more intuitive to work with for this application. The HSV values are then filtered for the desired colours (in this case a specific shade of yellow and blue) and morphological operations applied in an effort to reduce any noise

present in the image. Moments are then instantiated and defined and from this, the area and centroid co-ordinates of any detected objects can be obtained. These co-ordinates are then checked to see whether they lie within the center of the image and sets the corresponding booleans (`centre_check_body` and `centre_check_object`) accordingly. These booleans are used in conjunction with a check for area size so that the robot knows whether or not it has detected an object or a body straight in front of it, setting `object_detected` or `body_detected` as needed. Finally, a live feed of the original image and both filtered images are displayed onto windows.

The ability for the robot to turn 90 degrees is provided by the `turn90(bool right)` function, which takes in a boolean argument that dictates the direction of the turn (true for right, false for left). Essentially, this function operates on the basis of publishing velocity commands while checking whether the robot's odometry matches those of a calculated target value and stopping when it does. However, the odometry values are updated once per spin so for the function to work, it has to be designed to be called after every spin of the ROS node. Angles are returned from the odometry sensors inside a message as numerical values, ranging from 0 to -1 for a position on the right hand side and 0 to 1 for a position on the left hand side [Figure 3].

Initially, the robot determines its starting position by saving the current angle to two global variables, `current_angle` and `start_angle`. After this has been performed, the starting quadrant is determined based on the robot's starting position and saved as integer `decide` [Appendix 1]. From this starting quadrant, a target odometry value can be calculated and this is defined as global float `target`. The function then performs a check to see if the robot's current position is within an acceptable range of the target and if it is, sets global boolean `stop` to true. This range (defined by the global float `error`) acts like a buffer and is necessary in order to reduce error as the robot does not come to a stop instantaneously. Lastly, the function publishes a velocity command based both on the value of `stop` and the direction of the turn defined by the argument `right`. The velocity value is zero if `stop` is true.

The ability for the robot to turn 360 degrees is provided by the `turn360()` function. It operates in a similar fashion to `turn90(bool right)`, with the function designed to be called once per spin until the halt conditions have been met. Where this function differs from `turn90(bool right)` is that instead of 4 possible starting areas, it has 3 [Figure 4]. This is determined in the beginning of the function and saved as an integer `decide`.

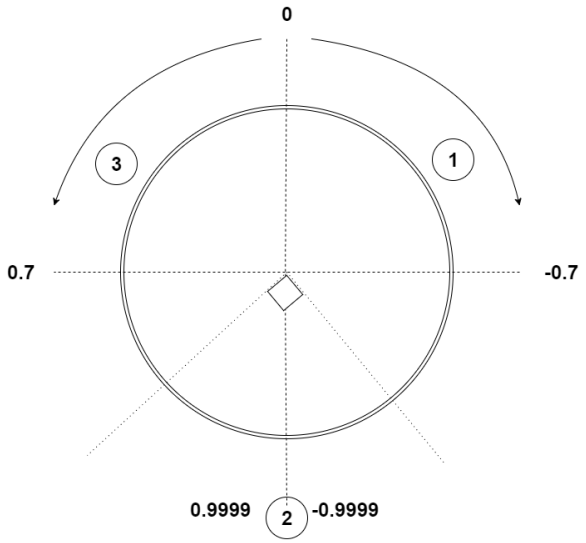


Figure 4 Possible starting areas for 360 degree turns

As the ideal target odometry for a 360 degree turn is the same as the starting odometry value, it is not possible to simply check if the robot is at the target as this would produce no movement. This function handles this issue by having a boolean, `past`, that sets to true once the robot has moved past the starting value. Once this has been set to true, it is possible to check the current position against the target position, setting boolean `halt` to true if the stopping condition has been met.

This function also checks to see whether a body or an object has been detected by the Kinect. For positive detections, it stores the robot's angular pose and x-y co-ordinates at that point into global arrays. This data is then processed at the end of the program to determine the actual co-ordinates of the detected objects and bodies, which will be explained further on in this section.

Finally, the switch statement publishes a non-zero velocity command on the basis that the stop conditions have not been met, otherwise it halts all turning.

Another core function is the ability to move linearly accurately, which is governed by the `travel(float distance)` function. This takes in a float argument that determines the distance the robot is to move in meters. This function operates by comparing the difference between the current linear pose of the robot and its starting position [Figure 5].

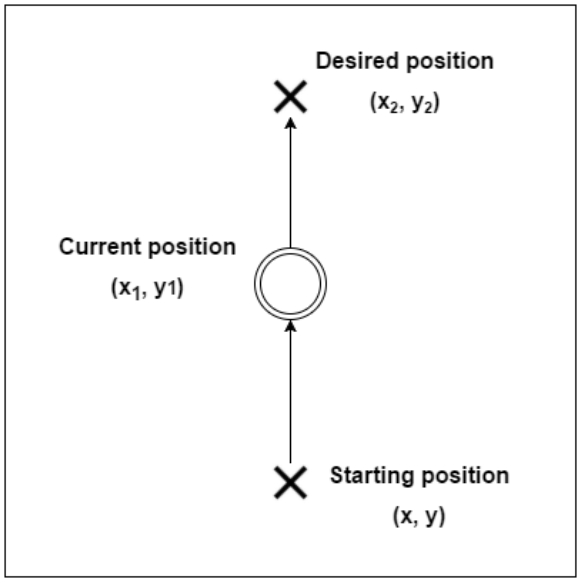


Figure 5 Travel function

This difference is tracked and saved as the float `travelled`. If this difference is less than the calculated target distance, defined by the float `desired`, then the robot has not travelled the required distance and calls the function `straight(bool direction)`, which publishes either a positive or negative linear velocity command depending on the value of the argument. However if the difference is greater than `desired`, the robot has travelled the required distance and the function sets all velocities to zero before publishing it.

The above function is utilized in `traveldecision(float distance_in)`, which is intended to check for obstacles in the path of the robot and maneuver appropriately. When the robot detects an obstacle in its path, it stops, turns 90 degrees and uses the laser scanner to check if there is still an obstacle blocking it. If the path is clear, it will attempt to maneuver around the obstacle, checking for any other obstacles on the way. If not, it will turn 180 degrees and perform similar checks and if clear, will attempt to maneuver around the other side of the obstacle.

The functions `createmodel()` and `createline()` were developed in order to visualize and track the odometry data coming from the sensors. `createmodel()` publishes a cylinder shaped marker over the marker node, with x-y co-ordinates corresponding to those of the robot. Similarly, `createline()` publishes a line over the marker node, which is used to show the path the robot has taken. This is achieved by publishing points with the x-y co-ordinates of the robot and linking them with a line. Both of these markers are displayable in Rviz, accessible in the "/base_laser_link" frame.

Finally, `process()` is called at the very end of the program, once the robot has finished executing the search pattern. As the name suggests, this function processes all of the data gathered during the program's execution. This is done sequentially from the first object/body detected to the last. Firstly, the

angular pose of the robot for the first body and object are converted to radians. Then the x-y co-ordinates of the body and object are calculated by adding the robot's linear pose at detection, with the corresponding x-y components of the object or body calculated by trigonometry [Figure 6]. Lastly, the angles are converted from radians into degrees.

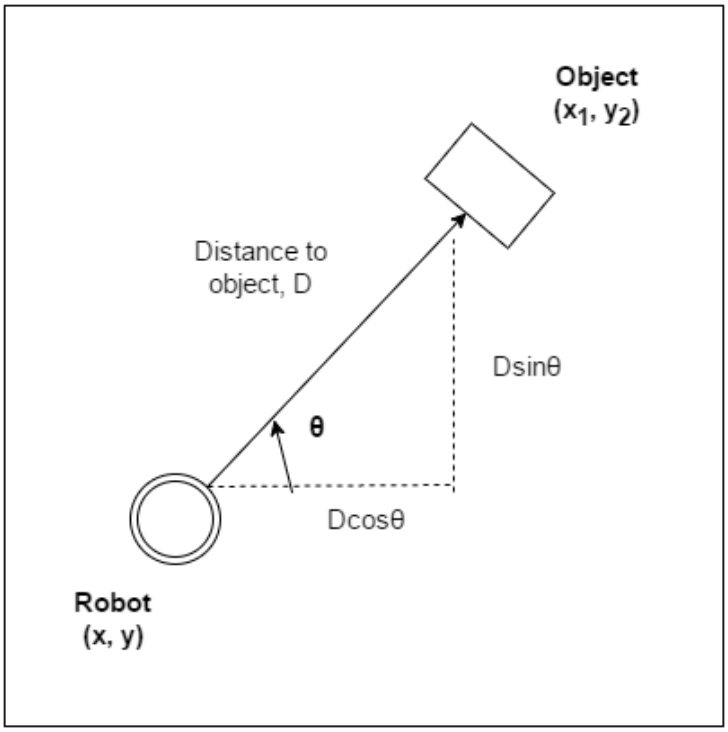


Figure 6 Determining co-ordinates

The majority of these lower level functions are utilized one way or another in `search()`, which is the function that chains together the movement and detection and executes the search pattern. As shown in Figure 2, this pattern takes the robot through the middle of the area of operation, performing three 360 degree scans along the way. `search()` chains together the movement by performing them sequentially through a switch statement, which is governed by the global integer `stage`. This variable is only incremented once the halt conditions of each function are met, meaning only one function within the switch statement is called at a time, enabling movement to be queued one after another.

4 PERFORMANCE

When it came to the demonstration, the robot did not perform all the tasks as optimally as it could have.

The proposed success criteria were that the robot's execution time was as low as possible, the number of collisions with obstacles was zero, the correct number of individuals detected and for the robot to not venture beyond the area of operation.

Where the robot performed well were with regards to execution time and boundary violations.

The search pattern was properly executed in well under two minutes and this was achieved by minimizing the number of movements performed. This also had the effect of reducing the total error in the movement functions caused by overshooting or undershooting the target location.

Initially, the limits of the operating area were coded into the program and the robot would know to stay away from the edges. However, it became evident that this was not needed as the robot would be spending the majority of its time in the middle of the search area, far away from the boundaries. This was shown during the demonstration, where the robot stayed well within the confines of the operating area.

However there were a few areas where performance could have been better.

Even though the search pattern and detection were technically accurate, the robot detected objects and bodies multiple times. This occurred not just on subsequent 360 degree scans detecting the same objects and bodies again, but occurred when detecting any object or body. This is because the speed of detection is much greater than the robot's physical angular velocity, leading to the object or body staying in the center long enough to trigger the detection multiple times. This resulted in many times more individuals being detected than there was present.

While performing the search, the robot collided with two obstacles. The first obstacle it collided with was due to the fact that it was a tiny distance out of range of the sensor while avoiding it. The second collision was owed to the robot overshooting a 360 degree turn and the obstacle was just within the subsequent target location.

Another area that was lacking was map building and visualization, neither of which were developed fully in time for the demo. The program was able to model the robot's current position and path taken by publishing its odometry values, which was displayable in Rviz. After executing the search pattern, it also has the co-ordinates of every body and object detected in the operating area relative to the robot's starting position. However, a way of plotting these points in Rviz was not developed.

5 CONCLUSION AND RECOMMENDATIONS

In conclusion, the robot in its current state is not an optimal solution for search and rescue. This is due to the software failing to completely meet some of the requirements outlined in the invitation to tender, such as building and storing a map, avoiding obstacles and detecting six individuals.

However, it does do many things correctly and expansions to the code can rectify many of the shortcomings present.

For example, the relative co-ordinates for the objects and bodies are readily available after the program has executed the search pattern. This can be plotted as markers in Rviz in a similar fashion to how the robot and the path taken are tracked currently.

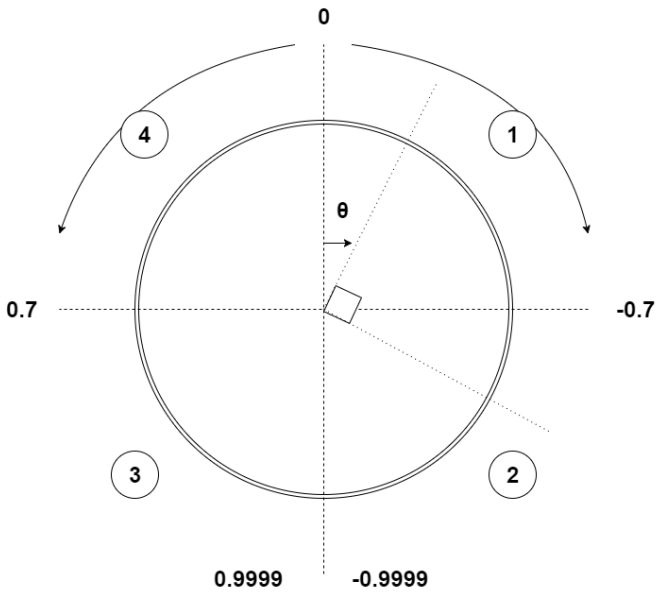
Simply doing this, however, would lead to upwards of thirty markers on the map where there are only six individuals present. Fortunately, owing to multiple detections, these markers will be clustered together and averaging the x-y co-ordinates of these clusters should produce an accurate co-ordinate for the detected object or body. `process()` would be an ideal function to implement this averaging and this should lead to six pairs of co-ordinates.

With regards to movement and odometry, this can be further improved by implementing a velocity function that varies the linear and angular velocity of the robot depending on its position. For example, if it were executing a 90 degree turn, it would slow down the closer it got to the target odometry value. This should have the effect of significantly minimizing error and prevent the robot from over-estimating or under-estimating any of its movements.

6 APPENDIX 1: EQUATIONS

Equations used for 90 degree turns, with starting position in:-

Quadrant 1:



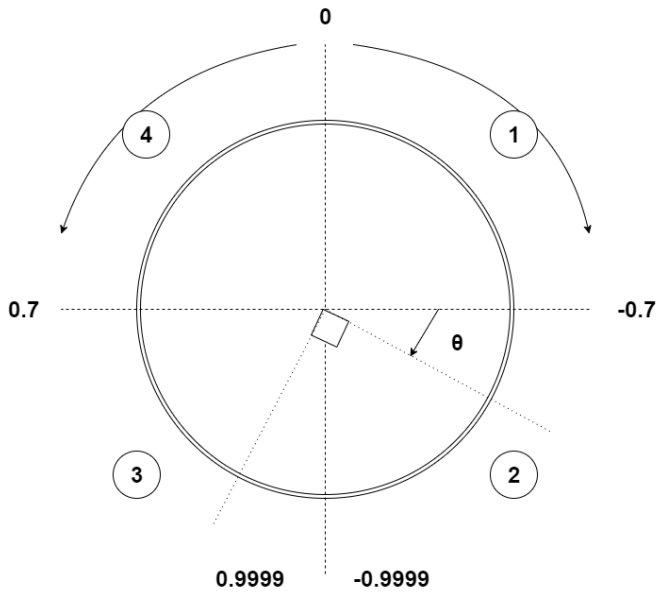
It can be seen that to turn 90 degrees, angle θ must be added onto relative east. However, the difference between relative east to relative south is 0.3 instead of 0.7, so a conversion ratio must be applied for any desired position within this quadrant.

Hence:

$$target\ odom. = \frac{start\ odom.}{\frac{0.7}{0.3}} - 0.7$$

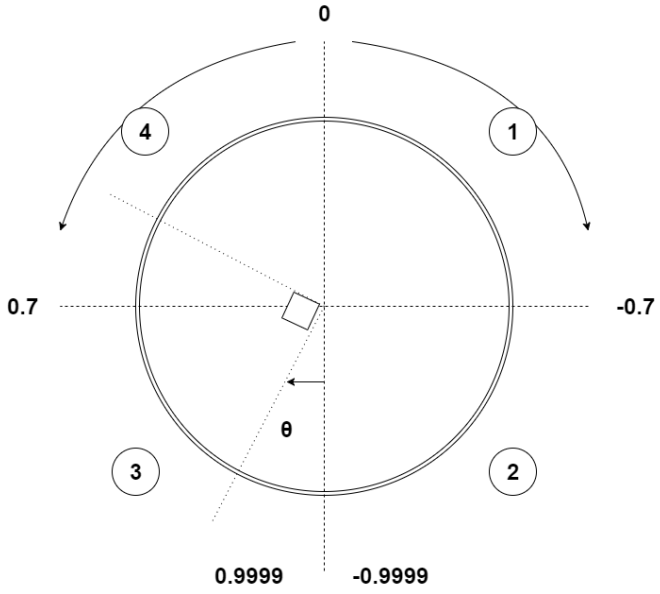
Helmi Fraser
H00152077

Quadrant 2:



$$target\ odom. = 1 + start\ odom. + 0.7$$

Quadrant 3:

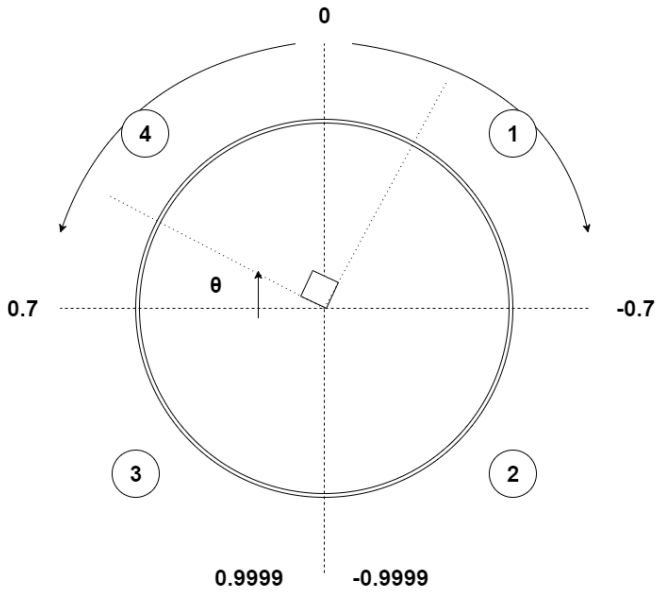


$$target\ odom. = 0.7\ - \frac{0.7}{0.3}(1\ -\ start\ odom.)$$

Here, the ratio must be applied in reverse.

Helmi Fraser
H00152077

Quadrant 4:



$$target\ odom. = start\ odom. - 0.7$$

7 APPENDIX 2: CONTRIBUTION

Job van Dieten: laser scanner functions, Rviz interfacing, higher level integration

Helmi Fraser: odometry, movement

Both: image processing, post processing