Contest 1 Report

MIE443 Mechatronics Systems: Design & Integration
Contest 1: Where am I? Autonomous Robot Search of an Environment

Due: February 13th, 2024

Team #4	
Team Member	**Student Number**
Henry (Hua Hao) Qi	1005758039
Harry Park	1005674405
Alastair Sim	1006460287
Yi Lian	1005709333

1.0 Problem Definition/Objective

The goal of MIE443: Contest 1 is to perform an autonomous robot search of an unknown environment. This search is conducted using a TurtleBot, which will drive around autonomously in the unknown environment while using the ROS gmapping libraries to generate a map using the sensory feedback from a Kinect sensor. The team is assigned to develop the exploration algorithm for the robot to navigate autonomously without human intervention. The exploration task is defined to be fixed timed, with a robot-based subject of action, and a movement goal of coverage. The fixed time constraint is placed as the robot must map the environment within a time limit. In a real-world scenario, this time constraint may be due to environmental or power restrictions. Although the task is fixed-timed, there is a preference for minimizing the mapping time. This is beneficial as the extra time could be used to account for unforeseen circumstances (ie. blockage). The subject of action is robot-based since the idea of the task is to have the robot place itself in the environment. Lastly, the movement goal is to maximize coverage since the robot should map as much of the unknown environment as possible.

1.1 Contest Requirements

The contest requirements are outlined as follows:

- The TurtleBot has a time limit of 8 minutes to both explore and map the environment.
- The robot must perform the overall task in autonomous mode. There will be no human intervention with the robot.
- The robot must use sensory feedback to navigate the environment. The robot cannot use a fixed sequence of movements without the help of sensors.
- There must be a speed limitation on the robot that will not allow it to go any faster than 0.25m/s when it is navigating the environment and 0.1m/s when it is close to obstacles such as walls. This is to ensure consistency in mapping.
- The contest environment will be contained within a 4.87x4.87 m^2 area with unknown static objects in the environment.

2.0 Strategy

Since the mapping environment is unknown until the day of the contest and a time constraint is placed for the mapping task, the team decided to develop an algorithm that allows the robot to quickly navigate the scene along with random movements to increase the coverage of the map. The overall algorithm is sequential as the code is divided into two halves. The first is a reactive wall-following approach, while the second involves deliberate turns to explore the center of the map. Additionally, the robot needs to avoid obstacles, which means it would need to redirect from its current position when the wall is too close, or if the bumpers hit an obstacle. To implement all of the above, the team developed an algorithm that defines 2 states of the robot: scanning and travelling. When the TurtleBot is in motion during the travel state or scanning state, it will subscribe to ROS topics to constantly check if any bumpers are hit, or if the laser sensors detect objects in front to avoid hitting obstacles.

2.1 Travelling State

During the travelling state, the robot will use a wall-following algorithm to guide the TurtleBot, where it turns to the left or right in small increments when it approaches too close to the wall. Although this does allow the robot to navigate through the maze and avoid obstacles, it can only produce the same sequence and may fall into a loop where it only reaches certain regions of the map. Hence, the team decided to implement additional actions with timeouts and loop counts to avoid such loopholes and add random rotations to increase the coverage while navigating.

2.1.1 Timeouts and Loop Counts in Travelling State

A time limit is set to the travel state so that after a certain amount of time has passed, the robot will halt the wall-following strategy and begin its attempt to navigate to the centre of the map. This is done by prompting the robot to make a 90-degree turn every 15 seconds if it is close to a wall while travelling. This allows the robot to scan any areas in the centre that it missed during its initial navigation.

Additionally, in case the robot gets stuck in a narrow pathway, the team added a loop count to limit the number of turns the robot can make during the travelling state. This allows the robot to escape dead-end areas where it cannot rely solely on its laser sensor readings.

2.2 Scanning State

The scanning state is implemented as a way to search for open areas in the environment when the robot is stuck in a narrow pathway or a corner of the environment. The idea of the scanning state is to scan 360 degrees around its surroundings while standing in place and redirect to the path with maximum clearance received from sensor feedback.

3.0 Robot Design

3.1 Sensory Design

The sensory design of the robot consists of proprioceptive and exteroceptive sensors. The proprioceptive sensors, which include the internal gyroscope and motor encoders, will be discussed in Section 3.1.1 and the exteroceptive sensors, which include the Kinect sensor and bumpers, will be discussed in Section 3.1.2.

3.1.1 Proprioceptive Sensors

Proprioceptive sensors are used to determine the internal state of the robot. They can help provide internal information about the robot, such as its orientation as well as linear and angular velocities. In our algorithm, we rely on using odometry to estimate the robot's position and orientation relative to a starting location. This starting point is given as an x and y coordinate as well as an orientation around the z-axis. While the x and y coordinate position of the robot is not used as often in our algorithm, the yaw is used extensively to track the turns that the robot makes. This is done using the internal gyroscope, which helps detect, measure, and maintain the angular motions of the robot. With the help of the gyroscope, we can specify a certain angle for the robot to turn, and determine when the robot is done turning by comparing the yaw of the initial and final positions of the robot. This can ensure that the intended turns for the robot are carried out accurately. As well, the gyroscope is also used to maintain the robot's angular velocity, which is crucial in both travelling and scanning. When travelling, the robot is programmed to turn at a low speed. This is because yaw measurements are taken every second, thus faster turns will lead to less accurate turns since it is easy to overshoot. In terms of scanning, the robot is programmed to turn 360 degrees slowly to make scans of its nearby surrounding environment and search for open areas to travel to. This is to maintain consistency and reduce errors in mapping while giving enough time to obtain the sensor distance corresponding to each measured yaw.

The second set of proprioceptive sensors used is the robot's motor encoders. A motor encoder is used to provide information on a rotary motor's speed and position. With the help of motor encoders, the robot's linear velocity can be specified and maintained. This is important as the linear velocity of the robot must be within the contest requirement, where it must not exceed 0.25 m/s when navigating and 0.1 m/s when close to obstacles. The robot is programmed to travel at a low speed, providing ample time for obstacle detection and avoidance while maintaining mapping quality.

3.1.2 Exteroceptive Sensors

Exteroceptive sensors are used to determine the external state of the robot. They can help provide external information about the environment, such as the layout of surrounding structures. In our algorithm, we rely on the Kinect depth sensor attached to the robot to visualize the unknown environment in which the robot is located. The Kinetic depth sensors use laser scanning by subscribing to the scan topic and retrieving laser distances. Using the 3D depth sensor, the robot has a 58-degree horizontal field of view and a nominal range of 0.8-3.5 meters, which is useful for both mapping and obstacle avoidance. The depth sensor acts similarly to eyes, where our algorithm uses the sensor feedback to determine if the robot needs to travel in another direction to avoid an obstacle and scan for which area has more open space for the robot to travel to. During the travel state, the robot is constantly using the depth sensor to scan the environment in front of it. If any structures are sensed, they will be mapped using the gmapping libraries. In the case that the obstacles are within a preset potential collision distance, our algorithm will determine the turn direction that will navigate the robot away from the obstacle, toward a more open area. This is done by dividing the laser scan into three sections: left, center, and right. By comparing the distances measured in the three sections, the direction with the farthest distance (equal to the most open area) is chosen as the new direction for the robot to travel. In addition to normal obstacle avoidance, if the robot gets stuck in a narrow pathway or a corner, the robot also uses a 360-degree scan to find a new direction of travel. These scans are also used to map any missing spots in the generated map.

On top of the Kinect sensor, the robot is equipped with three bumpers located on the left, center, and right sides of the base platform of the robot. These sensors help the robot detect any collisions with obstacles. The sensor feedback of the bumpers helps tell the robot if it has collided with any obstacles that may have been missed in its blind spot. Once the bumper has been pressed, the algorithm determines which direction the obstacle is in and moves the robot away from it.

3.2 Controller Design

To implement the strategy described in section 2.0, the controller was designed to operate the robot in two different main states: when the robot is moving and when it is stationary. When the robot is moving, it uses callback functions and custom functions explained in section 3.2.2 to update its parameters and check if its motion is complete, or if the robot must be stopped due to bumpers being triggered or the distance read from laser sensors being too close. Once the robot is stopped, it is directed to the stationary state, where it receives instructions for its next movement. With the help of custom functions, the robot can easily migrate from one state to another without being interrupted while in motion.

When the robot is stationary, instructions are given based on the robot's substate: either the scanning state or the travelling state. The travelling state is its default state, where it navigates through the environment by avoiding obstacles and following the edges of the obstacles according to its laser scan readings. The travelling state is divided into two parts in response to the total time elapsed. The first and second halves have the same functionality, with some modifications included for the second half such as timeouts and loop counts. Initially, the TurtleBot will travel based on a wall-following mechanism to sweep around the exterior environment. In the second half of the run, the robot will timeout every 15 seconds of moving in the same direction, and turn towards the centre of the map by reading its left and right laser sensors. It will also force the TurtleBot to travel through narrow paths if encountered. During any time of the run, if the loop count in travelling mode exceeds its limit, the TurtleBot is directed to the scanning state, where it completes a full rotational sweep of its surroundings and orients itself to travel in the most open position.

Since the robot is directed to carry out multiple movements during its stationary state, the team implemented a "step counter" that counts the steps of instructions to make sure each command was executed individually. The details of the algorithm are explained in the following sections along with a visual representation of the structure of the code in Figures 3.2.1 and 3.2.2.

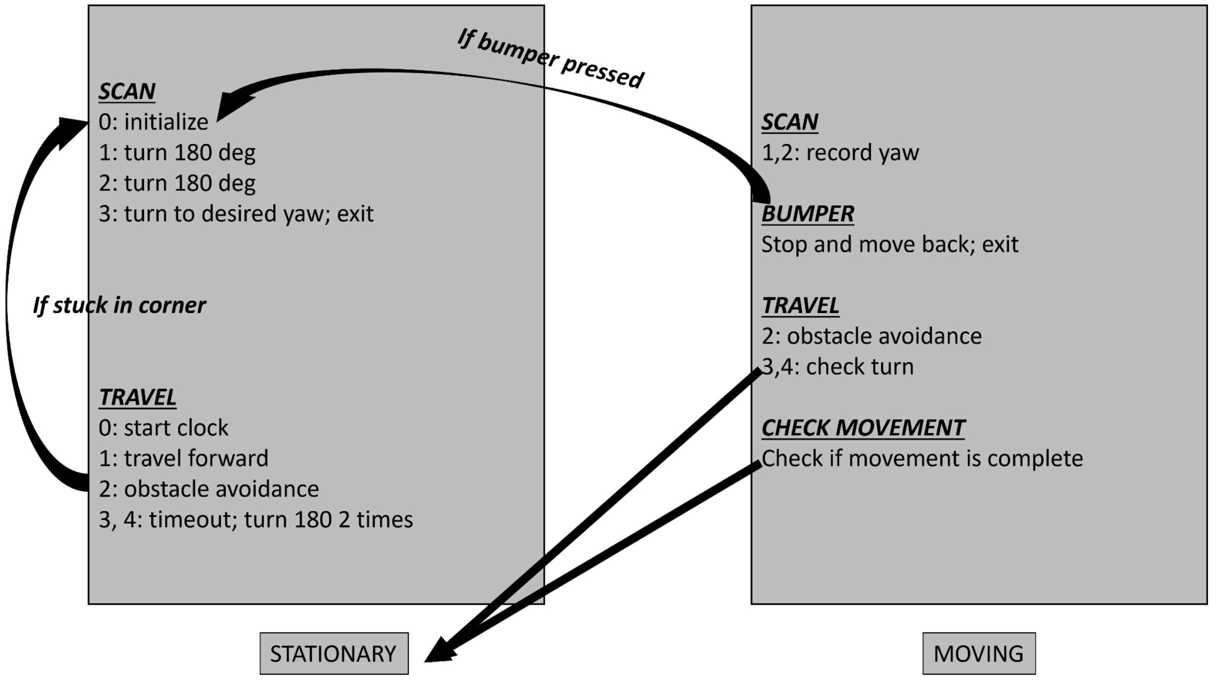

Figure 3.2.1: How Moving State Directs to Stationary State

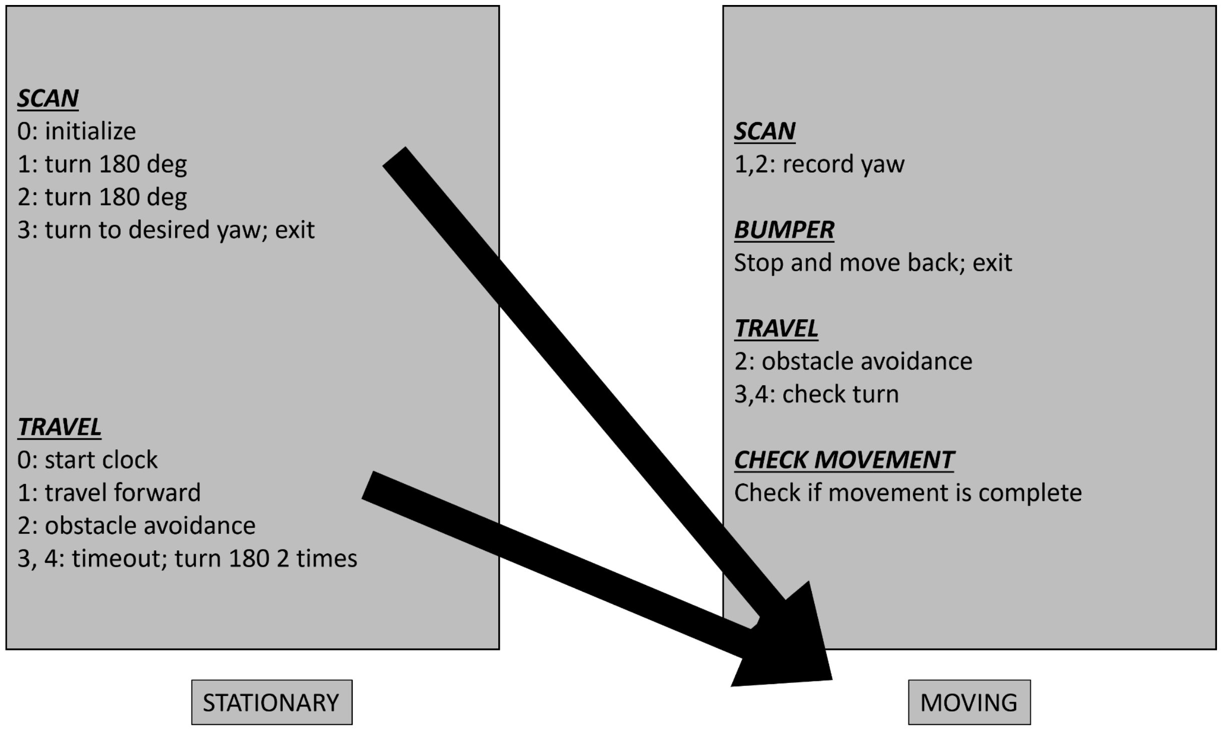

Figure 3.2.2: How Stationary State Directs to Moving State

3.2.1 Main Contest Code (contest1.cpp)

This is the main code that initializes ROS by subscribing to its odometry values, laser scan values, and bumper states, as well as publishing velocities. After initialization, a while loop is constructed to execute commands appropriately, as outlined in previous sections.

Upon startup, it executes *ros::spinOnce()* to update information on subscribed topics or to publish them. If the laser sensor readings are too close, it overrides its speed to 0.1m/s and increases the turning angle increments for turning in travelling mode. These adjustments were made to allow for more consistent mapping since the edges of the map tend to misalign. The travel time limit is also increased when close to obstacles, in order to compensate for its slow speed. The bumper states are also checked to see if any bumper was triggered. The last step of preliminary checking is to determine if the robot is moving or stationary according to its linear and angular velocities.

Initially, the robot's state is set to scanning mode, which allows it to scan 360 degrees around it using the custom *turnCCW()* function. Upon startup, however, the robot's yaw is set to 0, regardless of its actual orientation. To mitigate error from this misleading information, at step 0, no action is taken and increments the step by 1 to load the proper orientation. At steps 1 & 2, respectively, the robot is instructed to move 180 degrees. This is to ensure a full 360 sweep has been made, since turning 360 degrees is not ideal using the custom function. While it is turning, it enters the moving state, where it constantly checks if the turning is complete and records the yaw associated with the maximum open distance. To prevent the robot from moving in the same direction or turning directly back to where it came from, it will not record the yaw equal to the current orientation nor the orientation directly opposite to it. Once the scan is complete, it directs to the travelling mode after positioning itself to the desired orientation, decreasing the step to 0 again.

In the first half of the run, after the robot enters travelling mode, at step 0, it starts recording the time elapsed during the travel state. It then moves on to step 1, where it instructs to move forward at a speed predetermined at the beginning of the while loop. While it is moving, if any of the left, right, or centre laser readings fall below the stop limit, the robot will stop and enter step 2, where it adjusts its orientation by turning to the right or left in small increments. There is a limit on how many times it can turn, however, to ensure it does not get stuck on narrow corners. Once it exceeds the loop count limit, it exits the travel mode and redirects to scanning mode.

In the second half of the run, the TurtleBot will rotate 90 degrees away from the wall it is following. This is done by reading the left and right sensors every 15 seconds of moving in the same direction. If the left sensor reads a greater distance, it assumes the wall it is following is on the right side and turns 90 degrees counter-clockwise, and vice versa. Additionally, the robot will force movements towards narrow paths. This is achieved by forcing a movement forward in small increments if the front sensor readings exceed the left and right sensor readings, implying it is facing a narrow path in front of it.

When the robot is in motion, it constantly checks if the bumpers are pressed using the bumper callback function in *bumper.cpp*. If any of the bumpers have been pressed, the robot will move back 0.2m using the *moveBack()* function and redirect to the scanning state. This procedure is intended to be used only in situations where the laser sensors do not read properly and would need to redirect.

At the end of the while loop, the velocity parameters as well as elapsed time are updated.

3.2.2 Functions to Move and Turn (move.cpp and move.h)

The low-level control of our robot showcases how the robot executes the commands determined by the high-level control. This includes turning, forward and backward movement.

There are two turning functions implemented, one for clockwise (*turnCW()*) and one for counterclockwise (*turnCCW()*). The turning functions take in a target yaw and normalize it to ensure it remains in a 0-360 degree range. This avoids out-of-bounds errors and ensures smooth turning. The function keeps track of the remainder yaw after turns to ensure the robot reaches the desired orientation needed by the higher-level controls. For counter-clockwise turns the angular velocity value is set to a positive value and for clockwise turns the angular velocity is set to a negative value. Similar to turning, the forward (*moveFront()*) and backward ((*moveBack()*) movement controls take in a target distance and adjust the linear velocity accordingly.

The turning and moving functions call on correction functions as well. These correction functions (*checkMoveFront()*, *checkMoveBack(),* *checkTurnCCW()*, *checkTurnCW()*) continuously monitor the robot's orientation during rotation by checking if the TurtleBot has achieved its target yaw/ distance and adjusting rotation/ translation accordingly. In all of the above functions, linear and angular velocities, as well as the robot's current position are input as reference parameters so that they can be updated within the custom functions.

3.2.3 Laser Callback Function (laserCallback.cpp, laserCallback.h)

The laser callback function divides the field of view into 3 sections: centre, left, and right. It then calculates the minimum distance it reads from each of its sections, returning the values *minLaserDist*, *leftLaserDist*, and *rightLaserDist* respectively.

4.0 Future Recommendations (1 mark)

This section outlines recommendations for enhancements of the robot's exploration capabilities including frontier exploration, transforms, multi-threading, and PID controls. We would implement a frontier exploration algorithm that would be a separate state from our random exploration algorithm. The algorithm would use a cost map to define frontiers in the gmapping occupancy grid and we would need to test different ways of integrating the frontier algorithm with our random exploration algorithm. This would involve potentially using multi-threaded spinning in order to populate the cost map and either override the random control when a threshold has been reached in the cost map, overriding when passing by a potential frontier in order to save time or intermittent swaps between both algorithms. Another future implementation could be setting up transforms between the Kinect sensor, bumper sensors and kobuki bot. This would be beneficial to our robot allowing us to implement coordinate frames for our robot to understand where it is relative to other objects or locations. It would also allow for simplified sensor fusion by adding a relationship between the positions of the sensors and the robot. Furthermore, we could implement PID controls throughout our robot to ensure accuracy in our speed, heading and distance control. For example, PID could be used to maintain a speed based on differences between our desired and actual speed. As for headings and distance, they could be used to maintain a set distance from a wall or correct deviations in our movement from obstacles.

Our approach was to start with a simple autonomous drive control and then iterate by addressing one issue with our exploration at a time, eventually culminating in our random, wall-following approach. This approach was beneficial to learning about ROS and autonomous control and should be kept as our approach.

5.0 Appendix

Appendix A: Contribution Table

Section	**Subsection**	**Henry (Hua Hao) Qi**	**Harry Park**	**Alastair Sim**	**Yi Lian**
1.0					✔
2.0		✔	✔		
3.0	**3.1**	✔			✔
	3.2		✔	✔	
4.0				✔	
Code		✔	✔		✔

Appendix B: *contest1.cpp*

```
#include "globals.h"
#include "laserCallback.cpp"
#include "move.cpp"
#include "bumper.cpp"

bool local = true;

//// Initialize states & loop counts
bool isMoving = false;

int stepsCount = 0;
int subStepsCount = 0;
int travelLoop = 0;
uint64_t travelTimeLimit = 12;


//// Initialize variables for movement
float targetDist = 0.0;

float currentX = 0.0;
float currentY = 0.0;

float openYaw = 0.0;
float prevYaw = 0.0;

float turnAngle = 10.0;

float turtleSpeed = 0.0;
float turtleAngle = 0.0;

float maxLaserDist = 0.0; //variable to store recorded yaw during travel

//// Timer variables to record travel time
uint64_t travelElapsed = 0;
std::chrono::time_point<std::chrono::system_clock> travelStart;



int main(int argc, char **argv)
```

```
{
    ros::init(argc, argv, "image_listener");
    ros::NodeHandle nh;

    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);


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

    ros::Rate loop_rate(10);

    geometry_msgs::Twist vel;

    // Contest countdown timer

    std::chrono::time_point<std::chrono::system_clock> start;
    start = std::chrono::system_clock::now();
    uint64_t secondsElapsed = 0;

    // Initiallize angular and linear velocities
    float angular = 0.0;
    float linear = 0.0;

    // Initialize to scan state & substep 0
    stepsCount = SCAN_STEP;
    subStepsCount = 0;

    // Print statement to differentiate local and github repositories

    if (local) ROS_INFO("This is Local, not uploaded to github");
    else ROS_INFO("This is Oogway, uploaded to github");


    while(ros::ok() && secondsElapsed <= 480) {

        ros::spinOnce();
        ROS_INFO("Substep: %d", subStepsCount);
        ROS_INFO("FRONT: %g", minLaserDist);
        ROS_INFO("RIGHT-END: %g", rightLaserDist);
        ROS_INFO("LEFT-END: %g", leftLaserDist);

        //PRELIMINARY CHECKINGS

        //// Check if movement is complete

        isMoving = (angular != 0.0 || linear != 0.0);
        ROS_INFO("Angular speed: %f, Linear speed: %f", angular, linear);

        //// Check if bumper is hit

        bool anyBumperPressed = false;
        for (uint32_t b_idx = 0; b_idx < N_BUMPER; ++b_idx) {
                                        anyBumperPressed   |=   (bumper[b_idx]   ==
kobuki_msgs::BumperEvent::PRESSED);  //iterate  through  bumper,  check  if
pressed
        }

        ROS_INFO("Bumper is pressed: %d", anyBumperPressed);

        //// Check for speed limit

                if  (minLaserDist  <  slowDownLimit  &&((  leftLaserDist  <
slowDownLimit  ||  rightLaserDist  <  slowDownLimit)  &&  minLaserDist  <
slowDownLimit))
        {
            turtleSpeed = slowDown;
            turtleAngle = slowDownAngular;
            travelTimeLimit = 30;
            turnAngle = 20;

            ROS_INFO("SLOWING DOWN...");
        }

        else
        {
            turtleSpeed = normal;
            turtleAngle = normalAngular;
            travelTimeLimit = 15;
            turnAngle = 10;
            ROS_INFO("NORMAL SPEED");
        }



        //When Oogway is stationary, give directions
        if (!isMoving)
        {

            // SCANNING STEP
            if (stepsCount == SCAN_STEP)
            {

                ROS_INFO("SCANNING...");

                //skip first loop as yaw isn't updated yet
                if (subStepsCount == 0)
                {
                    subStepsCount++;
                }

                // initially turn 180 ccw
                else if (subStepsCount == 1)
                {
                    ROS_INFO("YAW:::: %f", yaw);

                    targetYaw = yaw +180;
                         turnCCW(targetYaw, angular, linear, remainingYaw,
turtleAngle);

                    subStepsCount++;

                }

                // turn another 180 ccw for 1 full loop
                else if (subStepsCount == 2)
                {
                    targetYaw = yaw +180;
                         turnCCW(targetYaw, angular, linear, remainingYaw,
turtleAngle);

                    subStepsCount++;
                }

                // direct to desired yaw and exit to travel mode
                else if (subStepsCount == 3)
                {
                    targetYaw = openYaw;

                    if (targetYaw > yaw)
                    {
                              if (yaw < 180 && yaw+360 - targetYaw < 180)
turnCW(targetYaw, angular, linear, remainingYaw, turtleAngle);
                                 else turnCCW(targetYaw, angular, linear,
remainingYaw, turtleAngle);
                    }

                    else
                    {
                             if (yaw >= 180 && targetYaw+360 - yaw < 180)
turnCCW(targetYaw, angular, linear, remainingYaw, turtleAngle);
                                  else turnCW(targetYaw, angular, linear,
remainingYaw, turtleAngle);
                    }



                    subStepsCount = 0;
                    stepsCount = TRAVEL_STEP;
                    maxLaserDist = 0.0;
                    ROS_INFO("Entering Travel mode");


                    travelLoop = 0;

                }
```

```
            }

            // TRAVELLING STEP
            else if (stepsCount == TRAVEL_STEP)
            {

                ROS_INFO("TRAVELLING (NOT MOVING YET)...");

                //Iniitate clock for timeout
                if (subStepsCount == 0)
                {
                    travelStart = std::chrono::system_clock::now();
                    subStepsCount++;
                }

                //give travel instructions - go straight
                else if (subStepsCount == 1)
                {
                    linear = turtleSpeed;
                    angular = 0.0;

                }

                // STOP LIMIT REACHED: instruct to turn left or right
                else if (subStepsCount == 2)
                {
                     if (travelLoop >= 6) //reduce turn increments if loop
count exceeds 5
                    {
                        ROS_INFO("Too much Gittering");
                        turnAngle = 10;

                        if (travelLoop >= travelLoopLimit)
                        {
                            subStepsCount = 0;
                            travelLoop = 0;
                            stepsCount = SCAN_STEP;
                            turnAngle = 20;
                        }
                    }
                        /// turn until clearance if travel loop limit not
reached

                    if (leftLaserDist > rightLaserDist)
                    {
                        targetYaw = yaw +turnAngle;
                         turnCCW(targetYaw, angular, linear, remainingYaw,
turtleAngle);
                        travelLoop++;

                    }

                    else
                    {
                        targetYaw = yaw -turnAngle;
                          turnCW(targetYaw, angular, linear, remainingYaw,
turtleAngle);
                        travelLoop++;
                    }

                    // move forward in small increments in narrow paths
                             if (secondsElapsed>240 && travelLoop >10 &&
leftLaserDist<minLaserDist && rightLaserDist<minLaserDist)
                    {
                        //break;
                        currentX = posX;
                        currentY = posY;
                        targetDist = (minLaserDist-stopLimit)/2;
                        moveFront(targetDist, currentX, currentY, angular,
linear, turtleSpeed);
                        travelLoop = 0;
                    }

                }

                // turn 90 towards the middle of the map after time limit
                else if (subStepsCount == 3)
                {
                    ROS_INFO("!!!!!!!!!!!!!!!TIME LIMIT!!!!!!!!!!!!!");

                    if (rightLaserDist > leftLaserDist)
                    {
                        targetYaw = yaw -90;
                          turnCW(targetYaw, angular, linear, remainingYaw,
turtleAngle);
                    }

                    else
                    {
                        targetYaw = yaw +90;
                         turnCCW(targetYaw, angular, linear, remainingYaw,
turtleAngle);


                    }

                    subStepsCount++;
                }




            }
        }

        //When bumper is hit while moving, stop and move back
        else if (anyBumperPressed)
            {
                ROS_INFO("BANG!!! Bumper Pressed");

                currentX = posX;
                currentY = posY;
                targetDist = 0.2;

                 moveBack(targetDist, currentX, currentY, angular, linear,
turtleSpeed);
                stepsCount = SCAN_STEP;
                subStepsCount = 0;

            }
        //Monitor Travelling
        else if (stepsCount == TRAVEL_STEP)
        {
            ROS_INFO("Travelling...");

            //if timeout reached & after 240s passed, turn 360
            if (travelElapsed > travelTimeLimit && secondsElapsed > 240)
                {
                    travelElapsed = 0.0;
                    subStepsCount = 3;
                    linear = 0.0;
                    angular = 0.0;
                }


              //complete scanning motion after transitioning from scanning
state
            if (subStepsCount == 0)
            {
                 if (angular > 0) checkTurnCCW(targetYaw, angular, linear,
remainingYaw, turtleAngle);
                else checkTurnCW(targetYaw, angular, linear, remainingYaw,
turtleAngle);
            }

            // Check for laserScan detection (obstacle avoidance)
            if (subStepsCount == 1)
            {
                      if  (leftLaserDist < slowDownLimit || minLaserDist <
slowDownLimit || rightLaserDist < slowDownLimit)
                {

                    angular = 0.0;
                    linear = turtleSpeed;
                }

                 if (leftLaserDist < stopLimit || minLaserDist < stopLimit
|| rightLaserDist < stopLimit)
                {
                    subStepsCount++;
                    angular = 0.0;
                    linear = 0.0;
                }
            }

            //obstacle avoidance - check if turning is complete
            else if (subStepsCount == 2)
            {

                if (angular != 0)
                {
                        if (angular > 0) checkTurnCCW(targetYaw, angular,
linear, remainingYaw, turtleAngle);
                            else checkTurnCW(targetYaw, angular, linear,
remainingYaw, turtleAngle);
                }

                // check movement for target distance
                if (linear !=0)
                {
                    //break;
                            checkMoveFront(targetDist, currentX, currentY,
angular, linear, turtleSpeed);
                }



                // Go back to travelling state if laser scan clears
                        if (leftLaserDist > clearLimit && minLaserDist >
clearLimit && rightLaserDist > clearLimit)
                {
                    angular = 0.0;
                    linear = 0.0;
                    subStepsCount--;
                    travelLoop = 0;
                    travelStart = std::chrono::system_clock::now();
                }
            }

            //Complete turning motion during 90 degree turns
            else if (subStepsCount == 4)
            {
                ROS_INFO("TURNING AFTER TIME LIMIT REACHED");
                   if (angular>0) checkTurnCCW(targetYaw, angular, linear,
remainingYaw, turtleAngle);
                else checkTurnCW(targetYaw, angular, linear, remainingYaw,
turtleAngle);

                if (angular==0.0) subStepsCount=0;
            }




        }

        //isMoving
        else
        {
            ROS_INFO("Moving...");

            //record yaw with most opening during 360 scan

            if (stepsCount == SCAN_STEP && subStepsCount != 0)
            {
                ROS_INFO("recording yaw...");
                      if (maxLaserDist < minLaserDist && minLaserDist !=
std::numeric_limits<float>::infinity())
                {
                    maxLaserDist = minLaserDist;
                    openYaw = yaw;
                    ROS_INFO("Max yaw is: %f", openYaw);
                }
            }

            //Checking if movement is done

            if (linear != 0)
            {
                    if  (linear  >  0)  checkMoveFront(targetDist,  currentX,
currentY, angular, linear, turtleSpeed);
                     else  checkMoveBack(targetDist,  currentX,  currentY,
angular, linear, turtleSpeed);
            }

            if (angular != 0)
            {
                 if  (angular  >  0)  checkTurnCCW(targetYaw,  angular,  linear,
remainingYaw, turtleAngle);
                else checkTurnCW(targetYaw,  angular,  linear,  remainingYaw,
turtleAngle);
            }
        }


        vel.linear.x = linear;
        vel.angular.z = angular;
        vel_pub.publish(vel);


        //Update timer of elapsed travel time
              if  (stepsCount  ==  TRAVEL_STEP  &&  subStepsCount  >  0  &&
subStepsCount < 3)
        {
                                                         travelElapsed     =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock
::now()-travelStart).count();
        }
        if (stepsCount == TRAVEL_STEP && subStepsCount == 2)
        {
            travelElapsed = 0;
        }

        //Update the timer
                                                      secondsElapsed       =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock
::now()-start).count();
        std::cout << "Time Elapsed:" << std:: endl;
        std::cout << secondsElapsed << std:: endl;
        std::cout << "Travel Time Elapsed:" << std:: endl;
        std::cout << travelElapsed << std::endl;
        std::cout << "Travel Loop:" << std:: endl;
        std::cout << travelLoop << std::endl;

        loop_rate.sleep();
        }

    return 0;
}
```

Appendix C : *move.cpp*

```
#include "move.h"

// Controls movement of robot including odometry callback function

float posX=0.0, posY=0.0, yaw=0.0;

float remainingYaw = std::numeric_limits<float>::infinity();
float targetYaw = std::numeric_limits<float>::infinity();

float remainingDist = std::numeric_limits<float>::infinity();
float dist = 0.0;




void odomCallback (const nav_msgs::Odometry::ConstPtr& msg)
{

    posX = msg->pose.pose.position.x;
    posY = msg->pose.pose.position.y;
    yaw = tf::getYaw(msg->pose.pose.orientation);
    if (yaw < 0) {
        yaw += 2*M_PI;
    }

    yaw = RAD2DEG(yaw);


}

//function to turn counterclockwise
void turnCCW (float& targetYaw, float& angular, float& linear, float&
remainingYaw, float& turtleAngle)
{
    //correct any out of bounds for targetYaw

    if (targetYaw >= 360)
    {
        targetYaw = targetYaw - 360;
    }
    else if (targetYaw <= 0)
    {
        targetYaw = targetYaw + 360;
    }


    if (yaw > targetYaw && abs(yaw-targetYaw) > 4)
    {
        targetYaw += 360;
    }

    remainingYaw = targetYaw - yaw; //initialize remainingYaw

    ///* logic to check ismovingis done
        if(remainingYaw <= 0.0 || abs(yaw-targetYaw) < 4) //stop once
remaining is less than 0
    {
        angular = 0.0;
        linear = 0.0;
    }

    else //keep rotating as long as there is remaining yaw
    {
        angular = turtleAngle;
        linear = 0.0;
    }

}

//function to turn clockwise
void turnCW (float& targetYaw, float& angular, float& linear, float&
remainingYaw, float& turtleAngle)
{

    //correct any out of bounds for targetYaw
    if (targetYaw >= 360)
    {
        targetYaw = targetYaw - 360;
    }
    else if (targetYaw <= 0)
    {
        targetYaw = targetYaw + 360;
    }


    if (yaw < targetYaw && abs(yaw-targetYaw) > 4)
    {
        targetYaw -= 360;
    }

    remainingYaw = yaw - targetYaw;



        if(remainingYaw  <=  0.0  ||  abs(yaw-targetYaw)  <  4)  //stop  once
remaining is less than 0
    {
        angular = 0.0;
        linear = 0.0;

    }

    else //keep rotating as long as there is remaining yaw
    {
        angular = -turtleAngle;
        linear = 0.0;
    }
}

void checkTurnCCW (float& targetYaw, float& angular, float& linear, float&
remainingYaw, float& turtleAngle)
{
    //correct any out of bounds for targetYaw

    if (targetYaw >= 360)
    {
        targetYaw = targetYaw - 360;
    }
    else if (targetYaw <= 0)
    {
        targetYaw = targetYaw + 360;
    }


       if  (yaw  >  targetYaw  &&  abs(yaw-targetYaw)  >  4)  //need  to  tune
threshold value 4
    {
        targetYaw += 360;
    }

    remainingYaw = targetYaw - yaw; //initialize remainingYaw

    ROS_INFO("Yaw: %f", yaw);
    ROS_INFO("Target Yaw: %f", targetYaw);
    ROS_INFO("Remaining Yaw: %f", remainingYaw);

    ///* logic to check ismovingis done
       if(remainingYaw  <=  0.0  ||  abs(yaw-targetYaw)  <  4)  //stop  once
remaining is less than 0
    {
        angular = 0.0;
        linear = 0.0;
        //ROS_INFO("Turning is false!");
        //ROS_INFO("Stopped turning...");

    }

}

void checkTurnCW (float& targetYaw, float& angular, float& linear, float&
remainingYaw, float& turtleAngle)
{

    //correct any out of bounds for targetYaw
    if (targetYaw >= 360)
    {
        targetYaw = targetYaw - 360;
    }
    else if (targetYaw <= 0)
    {
        targetYaw = targetYaw + 360;
    }



    if (yaw < targetYaw && abs(yaw-targetYaw) > 4)
    {
        targetYaw -= 360;
    }

    remainingYaw = yaw - targetYaw;

    ROS_INFO("Yaw: %f", yaw);
    ROS_INFO("Target Yaw: %f", targetYaw);
    ROS_INFO("Remaining Yaw: %f", remainingYaw);

       if(remainingYaw  <=  0.0  ||  abs(yaw-targetYaw)  <  4)  //stop  once
remaining is less than 0
    {
        angular = 0.0;
        linear = 0.0;

    }
}




/////MOVING
//functions to move forwards and backwards
void  moveFront  (float&  targetDist,  float&  currentX,  float&  currentY,
float& angular, float& linear, float& turtleSpeed)
{
    dist = sqrt(pow(currentX-posX, 2) + pow(currentY-posY, 2));

    remainingDist = targetDist - dist;

    if (remainingDist <= 0)
    {
        angular = 0.0;
        linear = 0.0;
        ROS_INFO("COMPLETED MOVING!!");
    }

    else
    {
        angular = 0.0;
        linear = turtleSpeed;

    }
}

void moveBack (float& targetDist, float& currentX, float& currentY, float&
angular, float& linear, float& turtleSpeed)
{
    dist = sqrt(pow(currentX-posX, 2) + pow(currentY-posY, 2));

    remainingDist = targetDist - dist;

    if (remainingDist <= 0)
    {
        angular = 0.0;
        linear = 0.0;
        ROS_INFO("COMPLETED MOVING!!");
    }

    else
    {
        angular = 0.0;
        linear = -turtleSpeed;

    }

}

//check functions to see if move completed - run while moving
void checkMoveFront (float& targetDist, float& currentX, float& currentY,
float& angular, float& linear, float& turtleSpeed)
{
    dist = sqrt(pow(currentX-posX, 2) + pow(currentY-posY, 2));

    remainingDist = targetDist - dist;

    if (remainingDist <= 0)
    {
        angular = 0.0;
        linear = 0.0;
        ROS_INFO("COMPLETED MOVING!!");
    }

    else
    {
        angular = 0.0;
        linear = turtleSpeed;
    }
}

void checkMoveBack (float& targetDist, float& currentX, float& currentY,
float& angular, float& linear, float& turtleSpeed)
{
    dist = sqrt(pow(currentX-posX, 2) + pow(currentY-posY, 2));

    remainingDist = targetDist - dist;

    if (remainingDist <= 0)
    {
        angular = 0.0;
        linear = 0.0;
        ROS_INFO("COMPLETED MOVING!!");
    }

    else
    {
        angular = 0.0;
        linear = -turtleSpeed;
    }
}
```

Appendix D: *laserCallback.cpp*

```
#include "laserCallback.h"

float minLaserDist = std::numeric_limits<float>::infinity();
float leftLaserDist = std::numeric_limits<float>::infinity();
float rightLaserDist = std::numeric_limits<float>::infinity();
int32_t nLasers=0, desiredNLasers=0, desiredAngle=15;


void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{

    minLaserDist = std::numeric_limits<float>::infinity();
    leftLaserDist = std::numeric_limits<float>::infinity();
    rightLaserDist = std::numeric_limits<float>::infinity();

    nLasers = (msg->angle_max-msg->angle_min)/ msg->angle_increment; //639
     desiredNLasers = DEG2RAD(desiredAngle)/msg->angle_increment; //5=53,
10=106, 15=159, 20=212

               //ROSTOPIC    INFO:    angle_max=0.524,    angle_min=-0.5215,
angle_increment=0.001636
    //angle can fit in between +-30 degrees
    //index 0 is right
    //index nLasers(max) is left



    // FRONT-FACING: 0 degrees, hence the midpoint of ranges' index

     if (desiredAngle*M_PI/180 < msg->angle_max && desiredAngle*M_PI/180 >
msg->angle_min){ //if desiredAngle is within max/min angles

        //CHECK CENTRE minDistance
            for (uint32_t laser_idx = nLasers/2-desiredNLasers; laser_idx
<nLasers/2 + desiredNLasers; ++laser_idx){ //index within desired range
            minLaserDist = std::min(minLaserDist, msg->ranges[laser_idx]);

        }
        //CHECK RIGHT maxDistance
                      for (uint32_t rLaser_idx = 0; rLaser_idx <
nLasers/2-desiredNLasers; ++rLaser_idx){
                              rightLaserDist = std::min(rightLaserDist,
msg->ranges[rLaser_idx]);
        }



        //CHECK LEFT maxDIstance
         for (uint32_t lLaser_idx = nLasers/2+desiredNLasers; lLaser_idx <
nLasers; ++lLaser_idx){
                                leftLaserDist = std::min(leftLaserDist,
msg->ranges[lLaser_idx]);
        }


    }
    else{
        for (uint32_t laser_idx=0; laser_idx<nLasers; ++laser_idx){
            minLaserDist = std::min(minLaserDist, msg->ranges[laser_idx]);
                     ROS_INFO("!!!OUT OF RANGE!!! - minLaserDist = %f",
minLaserDist);
        }
    }
}
```

Appendix E: *bumper.cpp*

```
#include "laserCallback.h"

float minLaserDist = std::numeric_limits<float>::infinity();
float leftLaserDist = std::numeric_limits<float>::infinity();
float rightLaserDist = std::numeric_limits<float>::infinity();
int32_t nLasers=0, desiredNLasers=0, desiredAngle=15;


void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{

    minLaserDist = std::numeric_limits<float>::infinity();
    leftLaserDist = std::numeric_limits<float>::infinity();
    rightLaserDist = std::numeric_limits<float>::infinity();

    nLasers = (msg->angle_max-msg->angle_min)/ msg->angle_increment; //639
     desiredNLasers = DEG2RAD(desiredAngle)/msg->angle_increment; //5=53,
10=106, 15=159, 20=212

               //ROSTOPIC    INFO:    angle_max=0.524,    angle_min=-0.5215,
angle_increment=0.001636
    //angle can fit in between +-30 degrees
    //index 0 is right
    //index nLasers(max) is left



    // FRONT-FACING: 0 degrees, hence the midpoint of ranges' index

     if (desiredAngle*M_PI/180 < msg->angle_max && desiredAngle*M_PI/180 >
msg->angle_min){ //if desiredAngle is within max/min angles

        //CHECK CENTRE minDistance
            for (uint32_t laser_idx = nLasers/2-desiredNLasers; laser_idx
<nLasers/2 + desiredNLasers; ++laser_idx){ //index within desired range
            minLaserDist = std::min(minLaserDist, msg->ranges[laser_idx]);

        }
        //CHECK RIGHT maxDistance
                      for   (uint32_t   rLaser_idx   =   0;   rLaser_idx   <
nLasers/2-desiredNLasers; ++rLaser_idx){
                              rightLaserDist   =   std::min(rightLaserDist,
msg->ranges[rLaser_idx]);
        }



        //CHECK LEFT maxDIstance
         for  (uint32_t lLaser_idx = nLasers/2+desiredNLasers; lLaser_idx <
nLasers; ++lLaser_idx){
                                leftLaserDist   =   std::min(leftLaserDist,
msg->ranges[lLaser_idx]);
        }


    }
    else{
        for (uint32_t laser_idx=0; laser_idx<nLasers; ++laser_idx){
            minLaserDist = std::min(minLaserDist, msg->ranges[laser_idx]);
                    ROS_INFO("!!!OUT  OF  RANGE!!!  -  minLaserDist  =  %f",
minLaserDist);
        }
    }
}
```

Appendix F: Header files

globals.h

```
#ifndef GLOBAL_HEADER
#define GLOBAL_HEADER

// Contains all global variables

//libraries needed

#include <ros/console.h>
#include "ros/ros.h"
#include <geometry_msgs/Twist.h>
#include <kobuki_msgs/BumperEvent.h>
#include <sensor_msgs/LaserScan.h>

#include <stdio.h>
#include <cmath>

#include <chrono>

#include <nav_msgs/Odometry.h>
#include <tf/transform_datatypes.h>

using namespace std;

// equations, definitions
#define N_BUMPER (3)
#define RAD2DEG(rad) ((rad)*180. /M_PI)
#define DEG2RAD(deg) ((deg)* M_PI /180.)


//global variables


const float slowDownLimit = 0.9;
const float stopLimit = 0.7;
const float clearLimit = 0.75;

//State variables: Determines which state turtlebot is in
```

```
const int TRAVEL_STEP = 1; //travel state
const int SCAN_STEP = 0; //scanning state
const int travelLoopLimit = 15; //gittering loop limit

extern bool isMoving; //boolean to determing if turtlebot is moving or not

extern int stepsCount; //sets state of turtlebot
extern int subStepsCount; //substates
extern int travelLoop; //loop count during travelling

extern uint64_t travelTimeLimit; //timeoutlimit for travelling state



//odometery variables
extern float posX, posY, yaw;

extern float targetDist; //store target distance for moving
extern float currentX; //record current position for moving
extern float currentY;

//moving variables
extern float angular, linear;
extern int turning;
extern bool isMoving;

const float normal=0.25, slowDown=0.1; //speed settings
const  float  normalAngular  =  M_PI/6,  slowDownAngular  =  M_PI/4;  //turn
faster when close to walls during scanning

extern float turtleSpeed; //speed override
extern float turtleAngle; //angular spped override

extern float openYaw; //openYaw for turning after scan
extern float prevYaw;

extern float turnAngle; //turn increment during travel
```

```
//bumper variables
extern uint8_t bumper[3];
extern bool anyBumperPressed;

extern uint8_t leftState, rightState, centerState;



//laser variables
extern float minLaserDist, leftLaserDist, rightLaserDist;
extern int32_t desiredAngle;
extern int32_t nLasers, desiredNLasers;

#endif
```

move.h

```
#ifndef MOVE_HEADER
#define MOVE_HEADER

#include "globals.h"

extern float targetYaw;
extern float remainingYaw;
extern float angular, linear;
extern bool isMoving;

extern float remainingDist;
extern float dist;


void odomCallback (const nav_msgs::Odometry::ConstPtr& msg);

//functions to turn
void turnCCW (float& targetYaw, float& angular, float& linear, float&
remainingYaw, float& turtleAngle);
void turnCW (float& targetYaw, float& angular, float& linear, float&
remainingYaw, float& turtleAngle);

//check functions to see if turn completed - run while moving
```

```
void checkTurnCCW (float& targetYaw, float& angular, float& linear, float&
remainingYaw, float& turtleAngle);
void checkTurnCW (float& targetYaw, float& angular, float& linear, float&
remainingYaw, float& turtleAngle);

//functions to move
void moveFront (float& targetDist, float& currentX, float& currentY,
float& angular, float& linear, float& turtleSpeed);
void moveBack (float& targetDist, float& currentX, float& currentY, float&
angular, float& linear, float&turtleSpeed);

//check functions to see if move completed - run while moving
void checkMoveFront (float& targetDist, float& currentX, float& currentY,
float& angular, float& linear, float&turtleSpeed);
void checkMoveBack (float& targetDist, float& currentX, float& currentY,
float& angular, float& linear, float& turtleSpeed);

#endif
```

laserCallback.h

```
#ifndef LASER_HEADER
#define LASER_HEADER

#include "globals.h"

// Laser related functions
void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg);

#endif
```

bumper.h

```
#ifndef BUMPER_HEADER
#define BUMPER_HEADER

#include "globals.h"

void bumperCallback(const kobuki_msgs::BumperEvent::ConstPtr& msg);
```

```
#endif
```