University of Toronto

MIE443: Autonomous Robot Search of an Environment

Group 4 Members:

Mariem Ahmed

Katrina Cecco

Kevin Han

Anh Kiet Nguyen

Wenhan Jiang

Date of Submission: Feb. 23rd, 2021

1. Problem Definition

1.1. Objective

The main objective of Contest 1 is to develop a robot exploration algorithm that can independently navigate the Gazebo environment. The robot will need to navigate and explore the environment within a timeframe [1]. The TurtleBot will generate its map using sensory information from the Kinect sensor and other sensors available by ROS Gmapping Package.

1.2. Requirements and Constraints

- The TurtleBot must perform all the tasks autonomously.
- The TurtleBot must have a speed limit of less than 0.25m/s when exploring the maze [1].
- The TurtleBot must have a speed limit of 0.1m/s when getting close to obstacles (such as walls) [1].
- The exploration algorithm should be built to be robust for unknown environments with static obstacles [1].
- The TurtleBot must independently navigate and map the environment of 6x6 m^2. An example environment is shown in Figure 1 [1].

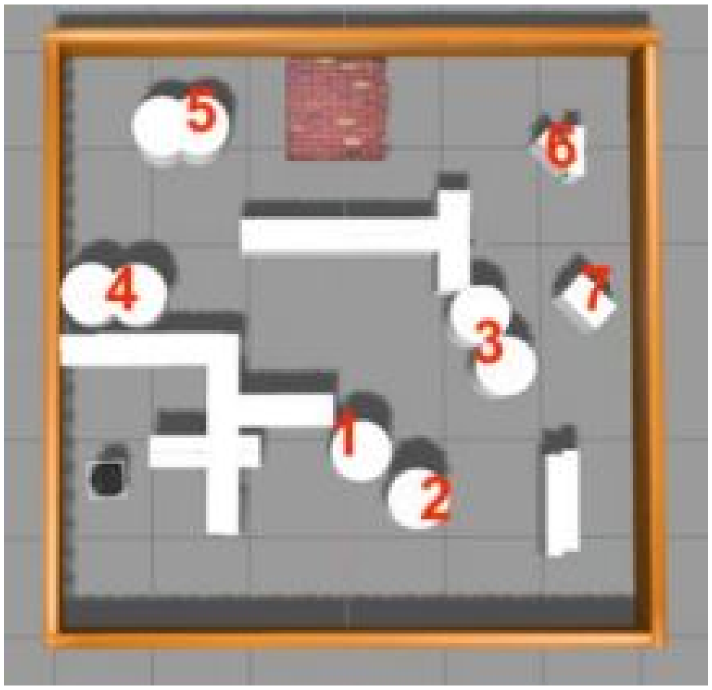

Figure 1: Example Environment [1]

2. Strategy

The main strategy that the team will use to get the robot explore the environment is open space explorer mode combined with wall-following mode.

In order to maximize the exploring area, the first step that our Turtlebot will implement is to do a 360° spin and use the laser scan sensor to find the direction with the maximum detected open distance. Next, the robot will rotate to that defined direction and start to move linearly until it reaches a wall or an obstacle.

Once the Turtlebot bumps into a wall, it will start its wall-following mode. By setting an upper and lower limit for the distance of the Turtlebot to the wall, our team programmed the movement of the Turtlebot such that it will turn away from the wall when the lower limit distance is crossed, turn toward the wall when the upper limit distance is surpassed, and move linearly when the Turtlebot stays in between these two limits. Although this will result in the zig-zag move of the robot and can be time-inefficient compared to a straight line moving parallel to the wall, our team chose this method due to its simplicity, and its potential to scan a wider area through the wall-following stage.

The robot alternates between advancing in the direction of the largest open space, and following a wall. Wall-following in particular has various modes that provide some variety of behaviour; these will be discussed in detail below.

3. Robot Design and Implementation

This section will explain in detail how the team implemented the sensors, high-level and low-level control for the robot design. The sensor control provides knowledge about the environment to help both the decision-making and control of the robot. The high-level control guarantees a robust and efficient navigation and mapping operation. The low-level control includes all primitive elements for supporting robot motion. Figure 2 shows an overview of the architecture of the robot design.

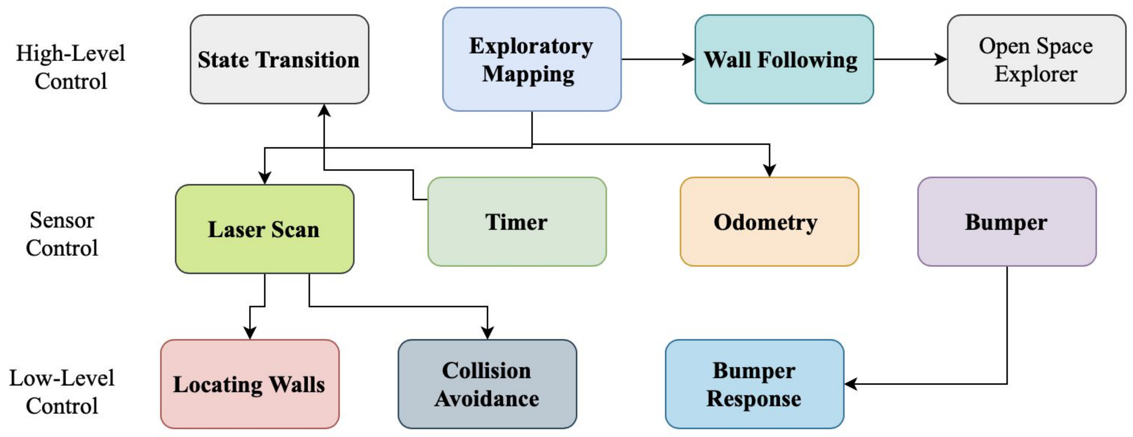

Figure 2: Robot Architecture's Overview

4. Sensory Design

The TurtleBot used in this course is divided into 2 main components: the Kobuki mobile base and the Simulated Microsoft Kinect 360. Each of these two contains multiple sensors. The Kobuki base includes 2x wheel drop sensors, 3x front bumpers, odometry, a gyroscope and 3 Cliff sensors, while the Kinect 360 is composed of an RGB camera and a depth sensor; both are shown in Figure 3. In Contest 1, the depth sensors, odometry, and the set of three bumpers are used for the navigation algorithms for the TurtleBot.

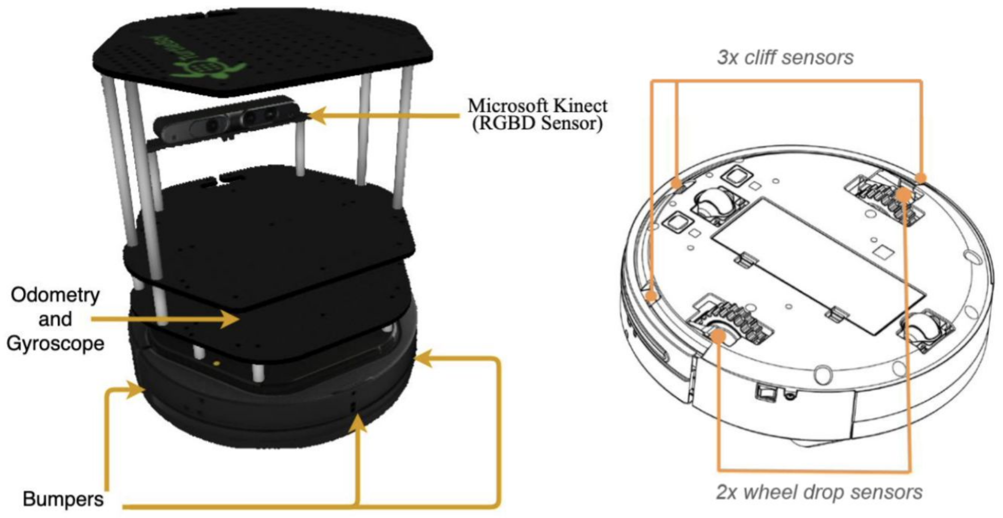

Figure 3: TurtleBot Hardware Components Overview [2]

4.1. Bumper Sensors

There are three bumper sensors located at the front of the base, one located on the left, front center, and right of the simulated Turtlebot, as shown in Figure 4. The bumper triggers when the TurtleBot runs into obstacles on the left, center or right side and provides information for the robot exploration algorithm to respond. The algorithm of our team for stimulating the bumpers will cause the robot to stop and try to free itself from an obstacle collision. Following this response, the robot moves into a new state, thereby starting a different behaviour from what led it into the wall in the first place. This is explained in depth in the Low-Level Controller Design.

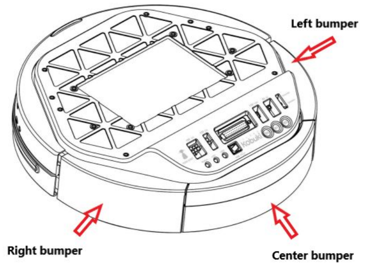

Figure 4: Kobuki base & locations of the three bumpers [2]

In the situation where the laser sensor cannot give us information, our team would need help from the three bumpers on the Kobuki base. The most important purpose of the bumpers is to identify the red brick, which cannot be scanned with the laser sensor, and help the Turtlebot avoid the red brick during the exploration.

The bumper is used to alert the Turtlebot that it hits a wall or an obstacle. Our team used the bumpers to detect a collision, as well as to program the robot to rotate away or back up itself from the site of collision depending on which bumper is pressed.

When the bumper is enabled, first we read which of the three bumpers is enabled and then behave as follows:

The Middle bumper is activated when we stop and then move back for a certain amount of distance to avoid any obstacles in front of the TurtleBot, until using the `collisionHandler()` function defined in the previous section to adjust the rotation and movement of the robot.

The Right and Left bumper when activated, we perform a motion to avoid any obstacles (such as walls) as the robot is too close to hit the wall and will move either right or left using the bumpers. First, the robot will step back a certain amount of distance of 0.1m and then the robot will turn away from obstacles clockwise/counter-clockwise. Before angling back again towards our initial direction of travel, we then step forward the same amount of distance.

4.2. Laser Scan Sensors

The depth sensor also referred to as the laser scan sensor is one of the two main parts of Microsoft Kinect 360, shown in Figure 5.

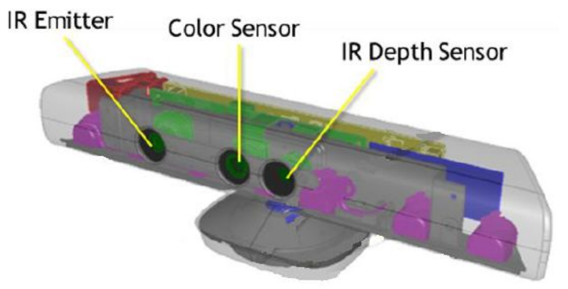

Figure 5: Microsoft Kinect 360 sensors [2]

We know that the laser sensor is not precise while travelling at high speeds, in particular when the TurtleBot spins. The speed of the robot is reduced to less than 0.25 m/s when exploring open surfaces and 0.1 m/s near obstacles and walls. The laser scan is used to store the distance to the wall in the forward direction and leftmost and rightmost lasers (`wallDist`). The minimum laser distance is also stored for four subsets of the entire laser scan range (e.g. `minRightDist`). These values are obtained by the laser scan function `(laserCallBack)`. Through the use of these three principles, we can align both our right and left sides' challenges during exploration.

From the 57^{o} laser range of the kinect sensor, our team chose 4 main divisions within this range to search for the minimum distance. The outer left laser range is at -28.5^{o} to -14.25^{o}, the middle left is -14.25^{o} to 0^{o}, the middle right is 0^{o} to 14.25^{o} and the outer right is 14.25^{o} to 28.5^{o} as shown in Figure 6 below. In addition, we also used the laser in the middle index (at 0^{o}) to measure the maximum and minimum distance in the forward direction. Most of the information that is used in our high-level controller design is drawn from the laser scan.

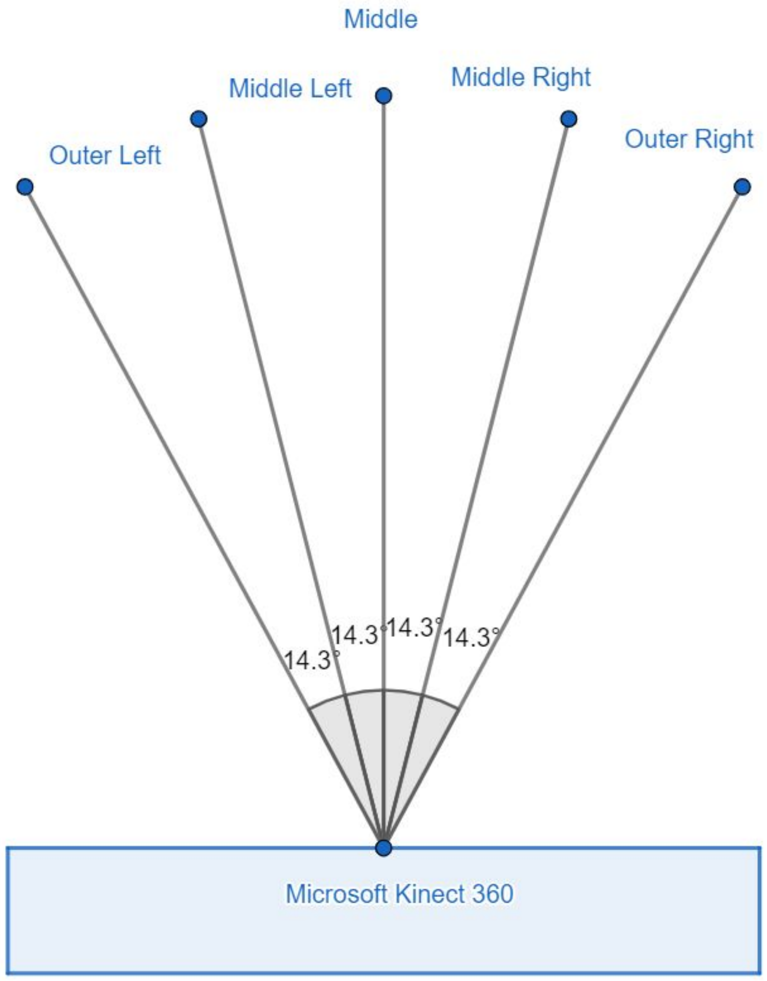

Figure 6: Laser Scan assignment

4.3. Odometry CallBack

The odometry will be used to approximate the location and orientation of the Turtlebot simulated in relation to a starter position in x and y position and to orientate the Turtlebot towards the *z* (upward) axis. We obtain our approximate x and y position from our odometry callback, as well as our actual yaw with regard to our starting position. Mainly, the yaw reading from odom callback is used to orient the robot to a desired heading, and to localize relative to walls in the environment.

5. Controller Design

5.1. High-level Controller Design

The high-level architecture is a behaviour-based architecture. A detailed discussion on each behaviour mode will be presented in section 5.2.

Within the main loop, prior to any behaviour, the robot updates its sensory information through `ros::spinOnce()` and the various callback functions (`laserCallback`, `odomCallback`, etc.). This is the main 'sensing' step of the robot. However, this is not the only time that sensory information is updated, as will be explained below.

At the level of the main loop, the behaviour-based architecture is implemented using a state machine. The robot's current state (or "behaviour mode") is stored as an integer `robotState`. The states include: spinning to locate maximum open distance, forward travel in direction of open distance, and wall following. For example, upon spawning, the robot's default state is an open-space explorer. When a certain behaviour is finished executing, the `robotState` variable is set to a new value to move the robot into another state. In general, most robot variables are maintained across states (e.g. `minLaserDist`).

This setup allows for a few benefits. Firstly, different behaviours can be programmed for each state and accessed repeatedly throughout the entire contest duration. Behaviours within states can be programmed in response to certain environmental conditions and are sometimes a function of the time elapsed (e.g. wall follower) to introduce some variety. From a development perspective, it is easier to debug behaviours when they can be called and tested independently of each other as different states. Using a state machine makes the program more modular and easier to add new behaviours by simply including a new `robotState` and appropriate transitions.

In addition to checking the `robotState` in each iteration of the main loop, there is always a check for collisions that supersedes the decision tree based on `robotState`. Collision checking can be considered yet another state, but is not encoded in the `robotState` variable for two reasons: first, that collisions serve as the primary transition between states; second, that the collision checks (which always result in the same sub-behaviours) do not have to be implemented individually in each state.

A downside of a behaviour-based state machine approach is that different state behaviours cannot be run concurrently. As a result, it can be difficult to account for cases whereby a robot might need to use information not typically available in that state. For example, if a collision has occurred with the front bumper, then the robot needs to back up. After a cycle or two of reversing, it is no longer touching the wall, so the bumper can no longer sense the wall. However, now must inform itself as to which way to spin. The solution is to perform additional calls to `rosSpinOnce()` and publish the velocity within certain behaviours, and not just at the end of the main loop.

Figure 7 is an overview of the High-Level Controller Design and each individual state will be further discussed in the Low-Level Controller Design (5.2.1).

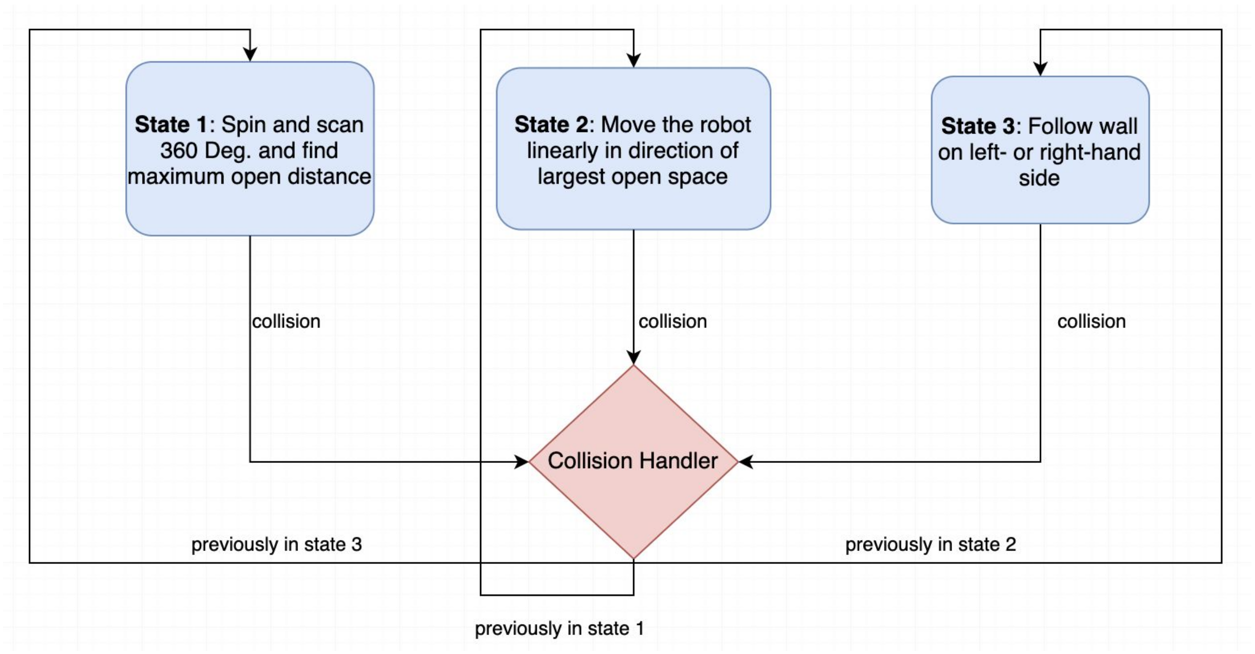
State 1: Spin and scan 360 Deg. and find maximum open distance
State 2: Move the robot linearly in direction of largest open space
State 3: Follow wall on left- or right-hand side
collision
collision
collision
Collision Handler
previously in state 3
previously in state 2
previously in state 1

Figure 7: Overview of the High-Level Controller Design

5.2. Low-level Controller Design

The low-level design of the robot enables the robot to detect the surrounding environment with sensory feedback: the robot travels within the maze and builds up the environment while following the walls and avoiding any obstacles.

We will give a more detailed explanation of how each part of the algorithm is implemented for states of the robot, collision avoidance and usage bumper.

5.2.1. States of the Robot

State 1: Locating the direction with maximum detected distance

When the Turtlebot first spawns on the map, it will do a full 360^o spin and scan the environment to seek open space. At the same time, it will record the maximum detected distance as well as the `yaw` angle (direction) that corresponds to that maximum distance. Once the robot finishes its full circle scan, it will turn to that `yaw` angle and is ready to move to the second state, which is to move linearly in that direction. This is shown in Figure 8. Following these function calls, the state is changed to state 2.

Two main functions are used to achieve our goal in these two initial states:

- **`scan360():`** this function is used to rotate the robot through 360 degrees and find the value of the maximum and minimum laser scans (`refMaxDistance, refMinDistance`). The `yaw` angle that corresponds to the maximum distances from the wall is stored as `maxDistAngle`, which is used in the **`turnToMaxDist()`** function. The `yaw` angle that corresponds to the minimum distances from the wall is stored as `minDistAngle`, which is used in the **`collisionHandler()`** function. Since the angular velocity is set to 22.5 degrees/sec for a better map scan resolution, we track the time of when the spin starts and halt the rotation once 16 seconds has passed, which is when the robot would have spun for a full 360 degrees at that speed. Overall, this function is used to rotate the robot in a full circle to provide the maximum and minimum distances of certain lasers or certain ranges of sections, which will be discussed in further detail below.

- **`turnToMaxDist():`** this function is used to orient the robot heading towards the `maxDistAngle` after the robot has scanned its environment in 360 degrees by calling **`scan360()`**. By rotating the robot until the current yaw matches the `maxDistAngle`, we are able to aim towards the maximum distances for the robot to move forward.

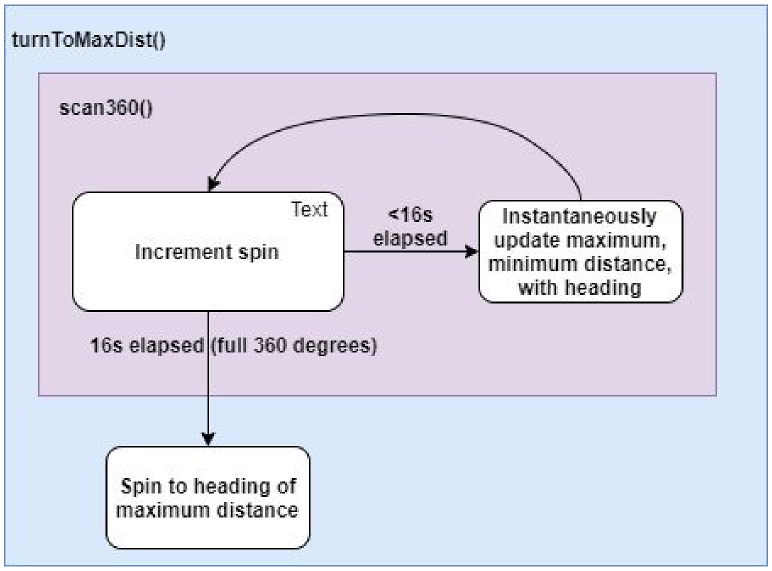
turnToMaxDist()
scan360()
Text
Increment spin
<16s elapsed
Instantaneously update maximum, minimum distance, with heading
16s elapsed (full 360 degrees)
Spin to heading of maximum distance

Figure 8: State 1 Flowchart

State 2: Moving linearly to the direction with a maximum distance

The speed used in this mode is chosen to be 0.25 m/s for a linear movement. In this state, the robot approaches a wall or obstacle within a certain distance and lock-on for the wall following. Distance feedback from the left and right lasers will be used to know which side of the robot is approaching the wall. Consequently, the linear speed will be reduced to 0.1m/s and an angular velocity will be initialized to prevent the robot from hitting the wall. In the case of red brick, where the laser scan cannot detect, or if the Turtlebot hit any objects on its way, the sensors from the three bumpers will be used. If the left bumper is pressed, the Turtlebot will turn left, and vice versa. If the front bumper is hit, the robot will back up by running at a negative velocity. The purpose of this state is to keep the Turtlebot at a certain distance from the wall and can proceed to the next state, which is the wall following.

State 3: Wall Follower Algorithm

Once the robot reaches within a clearance with a wall or objects, the wall-following state will start. The robot will begin to follow the left side wall or the right side wall depending on its current time condition. In particular, the robot will switch between the left and right wall following every 2.5 minutes to make sure the robot does not get stuck in the direction it is following.

When the robot is following a wall, an ‘if’ command will be used to keep the robot moving along the wall while maintaining a safe distance between the robot and the wall. Our team has chosen this clearance to be 0.8 m to 0.9 m. The depth sensor will be used to record the distance of the Turtlebot from the wall during this state.

In the case when the wall is on the right-hand side, the Turtlebot will firstly check the distance from the wall using the right side of the laser sensor. Two minimum distances will be recorded: one from the outermost right laser (R), and one from the middle right laser (MR). Four different cases will be taken into account: in good distance, too close to the wall, getting close to the wall and getting far from the wall.

The table below summarizes the four different cases for the robot behaviour.

Cases	**Distance**	**Control**	**Robot Behaviour**
In good distance	$0.8m \leq R \leq 0.9m$ & $MR > 0.8m$	$\omega = 0$ $V = 0.25$ m/s	Move linearly along this distance
Too close to wall	MR = infinity or MR 0.8m	$\omega = 30$ degrees/s $V = 0$	Rotate left only
Getting close to wall	($R < 0.8m$ & $MR > 0.8$) or (R = infinity & $MR > 0.8$)	$\omega = 10$ degrees/s $V = 0.15$ m/s	Slow down linear speed, Corner outwards to the left
Getting far from wall	$R > 0.9m$ and $MR > 0.8m$	$\omega = -10$ degrees/s $V = 0.15$ m/s	Slow down linear speed, Corner inwards to the right

In the case when our robot is going into a narrow pathway, we will reduce our clearance from (0.8m and 0.9m) to (0.5m to 0.6m).

As for the left wall following algorithm, the same strategy is applied, only with opposite signs of the rotating angle.

5.2.2. Collision Handler

In addition to the three main states above, the collision handler is a function that is activated when the Turtlebot collides with the environment.

The collision handler first reverses from the wall at 0.1m/s for one second, for a total of 10cm. This gives the Turtlebot enough clearance to turn and perform maneuvers without hitting the wall.

Then, the robot attempts to align itself parallel to the closest wall. It calculates the rotation angle required to achieve this (`parallelDiff`), then rotates through the desired angle. In the postcondition for this function call, the robot is aligned with the desired wall and ready to begin wall following or another behaviour. The robot state is then set according to the diagram in Figure 9.

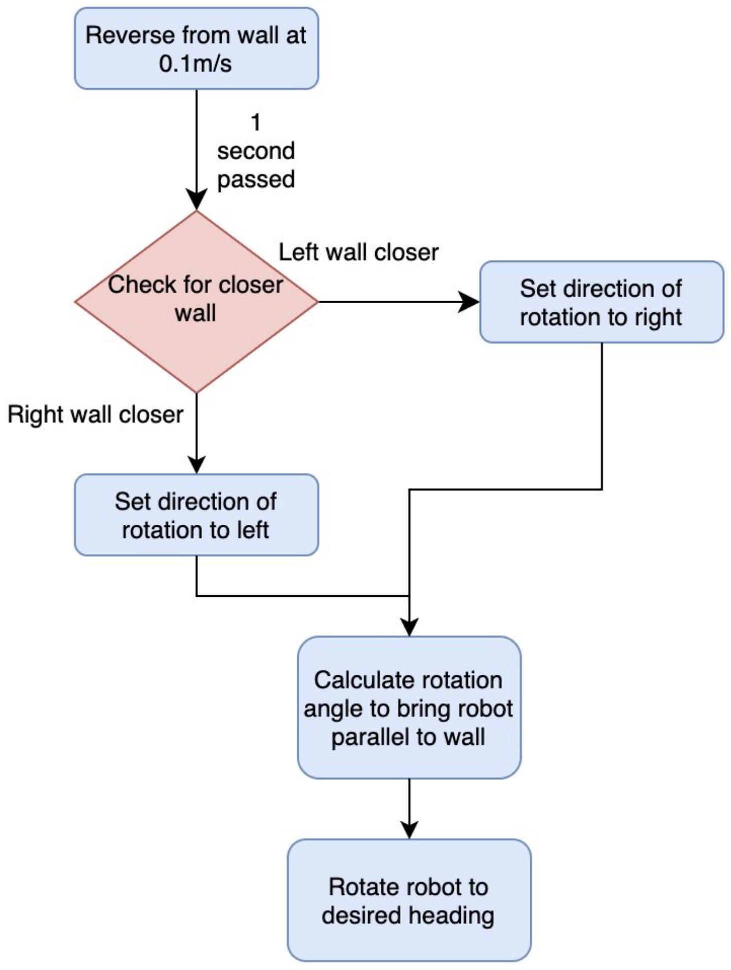
Reverse from wall at 0.1m/s
1 second passed
Check for closer wall
Left wall closer
Set direction of rotation to right
Right wall closer
Set direction of rotation to left
Calculate rotation angle to bring robot parallel to wall
Rotate robot to desired heading

Figure 9: Collision Handler Flowchart

6. Future Recommendation

If our team had more time to work on the project, we would implement a better algorithm for making the TurtleBot's movement be parallel to the wall as possible. We would use trigonometry to calculate the angle created by the wall and its current direction of the robot while moving. We would implement an algorithm to minimize the angle to zero when possible and the cases where the robot has to go back for turning away from any obstacles. The mechanisms we use to turn towards certain angles in collision handlers and turning towards max distance such as the time counter or margin of angle differences used to check come with some inaccuracies which may affect the efficiency of the robot movement.

There were also certain angles, corners, and obstacle dimensions that the wall follower could not easily get by, which would have to be optimized with the clearance ranges chosen for the functions. However, this was a challenging aspect to test and improve, since one slight change of certain factors would greatly impact the overall performance of the robot.

We also used switching between left and right wall followers to avoid getting stuck in certain areas, but this would take up a lot of the running time due to repeated collisions and rotations needed for new scans. In order to make this more efficient, we could build a smarter function which can detect which wall follower side will be best for the current situations. However there were too many failure cases and permutations that arose from solving this issue that made it complex to solve.

We could also consider implementing the random walk algorithm to determine the red brick wall, where the Turtlebot is moving linearly in a random direction. The algorithm will generate a random heading in the range of 0 to 2π radians (or 360^o). The robot will then rotate to that random direction and start to move linearly with 0.2m/s speed at a certain distance. The team would choose this path length to be 0.5m based on examining the map dimensions. When it reaches the end of the random trajectory, the Turtlebot will do a full 360^o rotation and spin, then scan. This random element would also be another mechanism that can be called periodically to prevent getting stuck in certain areas of the map.

Adjusting the level of clearances and speed combinations to make a smooth turn for any shaped obstacles was a challenge when testing. To solve this, we could have also looked to implement an algorithm for blind spot avoidance as shown in Figure 10, where the TurtleBot moves at a corner edge of a wall and the laser scan can not recognize that corner. To handle this case, the robot has to move forward while making random rotations to avoid collision.

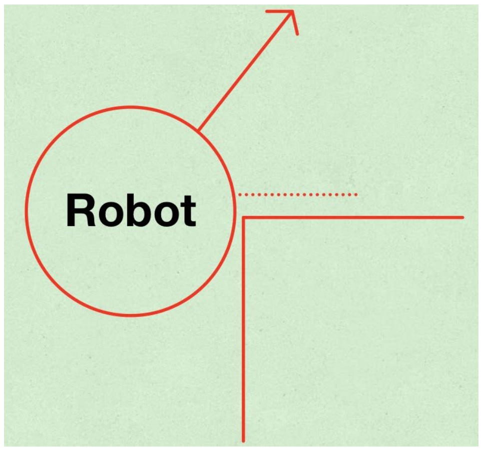

Figure 10: Collision due to blind spot

With our current design, the TurtleBot may not be able to explore the entire environment, as we are facing a problem with our approach for exploring. Therefore, there is no structured solution to successfully exploring the whole environment with the need to go around or re-exploring. A solution to this issue is to use the occupancy grid map included in the ROS packages to keep track of previously visited locations and, where possible, try to explore non-visited areas. There are also disadvantages, however, as it takes a non-negligible amount of time to generate and process the occupancy grid map and the position of the TurtleBot on the map may not be accurate, especially when errors accumulate as time goes by with more collisions. Consequently, this approach has limited resources and provides a balance between efficiency and completeness.

7. References

[1] MIE443: Contest 1 (2021). *Where am I? Autonomous Robot Search of an Environment*. Toronto: University of Toronto.

[2] *Kobuki User Guide*. 2017.

8. Appendix - C++ Code

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

// add new header files for subscribing to /odom
#include <nav_msgs/Odometry.h>
#include <tf/transform_datatypes.h>

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

#include <chrono>

// define conversion macros
#define N_BUMPER (3)
#define RAD2DEG(rad) ((rad) * 180./M_PI)
#define DEG2RAD(deg) ((deg) * M_PI / 180.)

float angular = 0.0;
float linear = 0.0;
float posX = 0.0, posY = 0.0, yaw = 0.0;
float margin = 0.02; // ~1 degree for margin of error

// define bumper param
uint8_t bumper[3] = {kobuki_msgs::BumperEvent::RELEASED,
kobuki_msgs::BumperEvent::RELEASED, kobuki_msgs::BumperEvent::RELEASED};
uint8_t robotState = 1; // 1 searching for maxDist wall, 2 go towards the wall,
3 following wall and finishing

// define laser param
float minLaserDist = std::numeric_limits<float>::infinity();
float minLeftDist = std::numeric_limits<float>::infinity();
float minLeftCentreDist = std::numeric_limits<float>::infinity();
float minRightCentreDist = std::numeric_limits<float>::infinity();
float minRightDist = std::numeric_limits<float>::infinity();
int32_t nLasers = 0, desiredNLasers = 0, desiredAngle = 1;
float maxLaserDist = 0.0, refMaxDist= 0.0, maxDistAngle = 0.0;
float refMinDist = std::numeric_limits<float>::infinity();
float minDistAngle = std::numeric_limits<float>::infinity();
float wallDist[3] = {0.0, 0.0, 0.0};
//float centerDist = 0.0;
float inf = std::numeric_limits<float>::infinity();

void bumperCallback(const kobuki_msgs::BumperEvent::ConstPtr& msg){
   //fill with your code
   bumper[msg->bumper] = msg->state;
}

void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg){
   //fill with your code
   minLaserDist = std::numeric_limits<float>::infinity();
   minLeftDist = std::numeric_limits<float>::infinity();
   minLeftCentreDist = std::numeric_limits<float>::infinity();
   minRightCentreDist = std::numeric_limits<float>::infinity();
   minRightDist = std::numeric_limits<float>::infinity();
   maxLaserDist = 0.0;

   nLasers = (msg->angle_max - msg->angle_min)/msg->angle_increment;
   desiredNLasers = DEG2RAD(desiredAngle)/msg->angle_increment;

   for (uint32_t laser_idx = 0; laser_idx < nLasers; ++laser_idx){
       minLaserDist = std::min(minLaserDist, msg->ranges[laser_idx]);
   }
   // rightmost, middle, leftmost laser distances
   wallDist[2] = msg->ranges[0];
   wallDist[1] = msg->ranges[nLasers/2];
   wallDist[0] = msg->ranges[nLasers - 1];

   for (uint32_t laser_idx = 0; laser_idx <= 159; ++laser_idx){
       minRightDist = std::min(minRightDist, msg->ranges[laser_idx]);
   }
   for (uint32_t laser_idx = 160; laser_idx <= 319; ++laser_idx){
       minRightCentreDist = std::min(minRightCentreDist,
msg->ranges[laser_idx]);
   }
   for (uint32_t laser_idx = 320; laser_idx <= 478; ++laser_idx){
       minLeftCentreDist = std::min(minLeftCentreDist, msg->ranges[laser_idx]);
   }
   for (uint32_t laser_idx = 479; laser_idx <= 639; ++laser_idx){
       minLeftDist = std::min(minLeftDist, msg->ranges[laser_idx]);
   }
}

// add odomCallback function
void odomCallback(const nav_msgs::Odometry::ConstPtr& msg){
   posX = msg->pose.pose.position.x; //set robot position
   posY = msg->pose.pose.position.y;
   yaw = tf::getYaw(msg->pose.pose.orientation); //convert quaternion to yaw
angle in radians
   // tf::getYaw(msg->pose.pose.orientation);

   // print robot position and orientation to screen
   //ROS_INFO("Position:(%f,%f) Orientation: %f rad or %f degrees.", posX,
posY, yaw, RAD2DEG(yaw));
}

void scan360(ros::Publisher vel_pub, geometry_msgs::Twist vel){
   //adjust positive - turn ccw
   //adjust negative - turn cw

   ros::spinOnce();
   std::chrono::time_point<std::chrono::system_clock> scanStart;
   scanStart = std::chrono::system_clock::now();
   uint64_t scanDuration = 0;
   angular = M_PI/8; // rotate for 16 seconds at this speed to perform 360 turn
   while (scanDuration < 16){
       scanDuration =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now
() - scanStart).count();
```

```
       //secondsElapsed =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now
()-start).count();
       vel.angular.z = angular;
       vel.linear.x = linear;
       vel_pub.publish(vel);

       ros::spinOnce();
       if (wallDist[1] > refMaxDist){
           maxDistAngle = yaw;
           refMaxDist = wallDist[1];
           ROS_INFO("maxDistUpdated: %f", wallDist[1]);
       }

       if (wallDist[1] < refMinDist){
           minDistAngle = yaw;
           refMinDist = wallDist[1];
           ROS_INFO("minDistUpdated: %f", wallDist[1]);
       }
   }
   angular = 0.0;
   linear = 0.0;
   vel.angular.z = angular;
   vel.linear.x = linear;
   vel_pub.publish(vel);
}

void turnToMaxDist(ros::Publisher vel_pub, geometry_msgs::Twist vel){
   // reset max distance reference for comparison
   refMaxDist = 0.0;
   // cannot call one 360 turn
   scan360(vel_pub, vel);
   ROS_INFO("maxDist: %f, maxDistAngle: %f", refMaxDist, maxDistAngle);
   angular = M_PI / 6;
   linear = 0.0;
   // turn until facing the desired angle
   while (abs(maxDistAngle - yaw) > margin && abs(maxDistAngle - yaw) < (2*M_PI
- margin)){
       vel.angular.z = angular;
       vel.linear.x = linear;
       vel_pub.publish(vel);
       ros::spinOnce();
   }
   angular = 0.0;
   linear = 0.0;
   vel.angular.z = angular;
   vel.linear.x = linear;
   vel_pub.publish(vel);
}

void collisionHandler(ros::Publisher vel_pub, geometry_msgs::Twist vel){
   //align front bumper to wall
   //front bumper aligned to wall - back up
   std::chrono::time_point<std::chrono::system_clock> collisionStart;
   collisionStart = std::chrono::system_clock::now();
   uint64_t collisionDuration = 0;

   ROS_INFO("collision! back up! minLaserDist: %f", minLaserDist);
   while (collisionDuration < 1){
       collisionDuration =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now
() - collisionStart).count();
       angular = 0.0;
       linear = -0.1;
       vel.angular.z = angular;
       vel.linear.x = linear;
       vel_pub.publish(vel);
       //ROS_INFO("startTime: %i, elapsed: %i", stateStartTime,
secondsElapsed);
   }
   angular = 0.0;
   linear = 0.0;
   vel.angular.z = angular;
   vel.linear.x = linear;
   vel_pub.publish(vel);
```

```
   // Re-orient the robot to be parallel (90 deg) from the min distance wall
after backing up
   refMinDist = std::numeric_limits<float>::infinity();
   scan360(vel_pub, vel);
   ROS_INFO("minDist: %f, minDistAngle: %f", refMinDist, minDistAngle);
   ros::spinOnce();

   float currentYaw = yaw;
   float parallelDiff = M_PI;
   if (currentYaw > minDistAngle){
       if (currentYaw - minDistAngle > M_PI){
           // case when left wall closer
           angular = -M_PI / 6;
           linear = 0.0;
           parallelDiff = M_PI + (currentYaw - minDistAngle - (2*M_PI));
       }
       else if (currentYaw - minDistAngle < M_PI){
           // case when right wall closer
           angular = M_PI / 6;
           linear = 0.0;
           parallelDiff = M_PI - currentYaw - minDistAngle;
       }
   }

   else if (minDistAngle > currentYaw){
       if (minDistAngle - currentYaw < M_PI){
           // case when left wall closer
           angular = -M_PI / 6;
           linear = 0.0;
           parallelDiff = M_PI - minDistAngle - currentYaw;
       }
       else if (minDistAngle - currentYaw > M_PI){
           // case when right wall closer
           angular = M_PI / 6;
           linear = 0.0;
           parallelDiff = M_PI + (minDistAngle - currentYaw - (2*M_PI));
       }
   }
   float endRotateTime = (parallelDiff / angular);

   std::chrono::time_point<std::chrono::system_clock> parallelTurnStart;
   parallelTurnStart = std::chrono::system_clock::now();
   uint64_t parallelTurnDuration = 0;
   while (parallelTurnDuration < endRotateTime){
       parallelTurnDuration =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now
() - parallelTurnStart).count();
       //secondsElapsed =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now
()-start).count();
       vel.angular.z = angular;
       vel.linear.x = linear;
       vel_pub.publish(vel);
   }
   angular = 0.0;
   linear = 0.0;
   vel.angular.z = angular;
   vel.linear.x = linear;
   vel_pub.publish(vel);
}

void leftWallFollower(ros::Publisher vel_pub, geometry_msgs::Twist vel){
   ros::spinOnce();
   float inf = std::numeric_limits<float>::infinity();
   float gap1 = 0.9;
   float gap2 = 0.8;
   if (minRightDist < 0.2){
       // narrow corridor or corner
       //ROS_INFO("gap narrow");
       gap1 = 0.6;
       gap2 = 0.5;
   }
   if (minLeftDist >= gap2 && minLeftDist <= gap1 && minLeftCentreDist > gap2){
       // maintain distance along the straight wall
       linear = 0.25;
       angular = 0.0;
   }
   else if (minLeftCentreDist <= gap2 || minLeftCentreDist == inf){
       // inf when too close, only rotate around
       linear = 0;
       angular = -M_PI/6;
   }
   // cornering inwards
   else if (minLeftDist > gap1 && minLeftCentreDist > gap2){
       linear = 0.15;
       angular = M_PI/18;
   }
   // cornering outwards
   else if ((minLeftCentreDist > gap2 && minLeftDist < gap2) ||
(minLeftCentreDist > gap2 && minLeftDist == inf)){
       linear = 0.15;
       angular = -M_PI/18;
   }
}

void rightWallFollower(ros::Publisher vel_pub, geometry_msgs::Twist vel){
   ros::spinOnce();
   float inf = std::numeric_limits<float>::infinity();
   float gap1 = 0.9;
   float gap2 = 0.8;
   if (minLeftDist < 0.2){
       // narrow corridor or corner
       //ROS_INFO("gap narrow");
       gap1 = 0.6;
       gap2 = 0.5;
   }
   if (minRightDist >= gap2 && minRightDist <= gap1 && minRightCentreDist >
gap2){
       // maintain distance along the straight wall
       linear = 0.25;
       angular = 0.0;
   }
   else if (minRightCentreDist <= gap2 || minRightCentreDist == inf){
       // inf when too close, only rotate around
```

```
        linear = 0;
        angular = M_PI/6;
    }
    // cornering inwards
    else if (minRightDist > gap1 && minRightCentreDist > gap2){
        linear = 0.15;
        angular = -M_PI/18;
    }
    // cornering outwards
    else if ((minRightCentreDist > gap2 && minRightDist < gap2) ||
(minRightCentreDist > gap2 && minRightDist == inf)){
        linear = 0.15;
        angular = M_PI/18;
    }
}

int main(int argc, char **argv){
    ros::init(argc, argv, "image_listener");
    ros::NodeHandle nh;

    //Node subscribe to bumper/scan topic
    ros::Subscriber bumper_sub = nh.subscribe("mobile_base/events/bumper", 10,
&bumperCallback);
    ros::Subscriber laser_sub = nh.subscribe("scan", 10, &laserCallback);
    //Publish a velocity command
    ros::Publisher vel_pub =
nh.advertise<geometry_msgs::Twist>("cmd_vel_mux/input/teleop", 1);

    //  ros::Rate loop_rate(10);

    // add odom subscriber
    ros::Subscriber odom = nh.subscribe("odom", 1, &odomCallback);

    geometry_msgs::Twist vel;

    // contest count down timer
    std::chrono::time_point<std::chrono::system_clock> start;
    start = std::chrono::system_clock::now();
```

```
   uint64_t secondsElapsed = 0;

   while(ros::ok() && secondsElapsed <= 900) {
       // while ros is running and under 15 minutes
       // spin once = messages received from subsribed topics are processed
       ros::spinOnce();

       //ROS_INFO("Position:(%f,%f), Orientation: %f degrees, Range: %f", posX,
posY, RAD2DEG(yaw), minLaserDist);
       //Check for any bumpers hit / run into wall / stuck -> turn to the right
if stuck; if nothing in front then advance
       bool 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);
       }

       if (any_bumper_pressed){
           ROS_INFO("collision start");
           collisionHandler(vel_pub, vel);
           if (robotState == 1){
               robotState = 2;
           }
           else if (robotState == 2){
               robotState = 3;
           }
           else if (robotState == 3){
               robotState = 1;
           }
           ROS_INFO("collision end");
           //turnToMaxDist(vel_pub, vel, M_PI/2);
           //ROS_INFO("3");
           //break;
       }

       // First state: check surroundings, search for open space, head towards
maxLaserDist
       else if (robotState == 1){
           // once finished, turn to the yaw angle that gave maxLaserDist
           turnToMaxDist(vel_pub, vel);
           // ROS_INFO("maxDistAngle %f, maxLaserDist %f", maxDistAngle, 
maxLaserDist);
           robotState = 2;
           ROS_INFO("state 1 complete");
       }

       // Second state: go towards the maxDist wall, if bumper pressed: 
collisionHandler then wall following as shown above
       else if (robotState == 2){
           //ROS_INFO("state 2 start");
           linear = 0.2;
           angular = 0.0;
       }

       // Third state: follow the wall by realigning the robot heading to be 
parallel to the wall (done in collisionHandler)
       else if (robotState == 3){
           if (secondsElapsed < 150 || (secondsElapsed > 300 && secondsElapsed 
< 450) || (secondsElapsed > 600 && secondsElapsed < 750)){
               ROS_INFO("now following left");
               leftWallFollower(vel_pub, vel);
           }
           else if ((secondsElapsed > 150 && secondsElapsed < 300) || 
(secondsElapsed > 450 && secondsElapsed < 600) || (secondsElapsed > 750 && 
secondsElapsed < 900)){
               ROS_INFO("now following right");
               rightWallFollower(vel_pub, vel);
           }
           // switch between which walls to follow to ensure it doesn't get 
stuck somewhere due to direction of following
       }
       // publish info to the simulated turtlebot: /teleop topic
       vel.angular.z = angular;
       vel.linear.x = linear;
       vel_pub.publish(vel);
       // The last thing to do is to update the timer.
       secondsElapsed =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now
()-start).count();
       //loop_rate.sleep();
   }
   ROS_INFO("Time's up!");
   return 0;
}
```

9. Contribution Table

		Katrina	Kevin	Anh	Mariem	Wenhan
Code	`laserCallBack`	AD, WC, TC	AD, WC, TC			
	`scan360`	TC	AD, WC, TC			
	`turnToMaxDist`	TC	AD, WC, TC			
	`collisionHandler`	AD, WC, TC	AD, WC, TC			
	`left/rightWallFollower`	AD	AD, WC, TC	AD	AD	AD
	`Main` (high-level controller)	AD, WC, TC	AD, WC, TC			
Report	Problem Objective				RD	
	Strategy			RD, MR		
	Robot Design and Implementation				RD, MR	
	Sensory Design	MR		RD, MR	RD, MR	
	High-Level Controller Design	RD, MR	MR			
	Low-Level Controller Design	MR	MR	RD, MR	RD	RD
	Future Recommendation		MR	RD	RD, MR	RD, MR
	Overall Document	ET, FP	ET, FP	ET, FP	ET, FP	ET, FP

RS – research **RD** – wrote first draft **MR** – major revision **ET** – edited for grammar and spelling **FP** – final read **WC** - Written Code **TC** - Testing Code **AD** - Algorithmic Design