EE4308 Autonomous Robot Systems

Semester 2, 2022/2023

Project 1 - Report

Name	Matriculation Number
Thet Ke Min, Sonia	A0221084R
Foo Fang Kiang	A0222770L
Stevanus Williem	A0221237R
Tran Duy Anh	A0200724W

1. Introduction 3
2. Project Tasks 3
2.1. Coupling Angular Error with Linear Error 4
2.2. Bi-Directional Motion in Motion Controller 6
2.3. Map Drifting (Goal in Inflation Zones) 7
2.4. Collection of Data 9
2.4.1. Tuning of Motion Filter Weights 9
2.4.2. Tuning of PD Gains 10
2.5. Cubic Hermite Trajectory Generation 12
3. Debug & Fix Any Errors 12
4. Conclusion 13
5. Appendix A (code for main.cpp) 14
6. Appendix B (code for move.cpp) 21
7. Appendix C (code for planner.cpp) 28
8. Appendix D (code for trajectory.cpp) 34

1. Introduction

The purpose of Project 1 is to autonomously move Turtlebot3 Burger on a flat area with various obstacles to reach three coordinated targets.

Figure 1: Project 1's map

Our design includes a bi-directional motion controller, a mechanism to find the closest free space when the goal is in the inflation zone using Dijkstra, Cubic Hermit trajectory generation algorithm, and post-process mechanism.

To achieve this, our team has incorporated what we have learnt from Lab 1 & Lab 2. Various tools have been used to aid our journey in tackling certain tasks as listed in the following section – Project Tasks. Overall, we have taken a lot of steps to ensure the robot navigates smoothly within the given duration of two minutes.

2. Project Tasks

In the following, we have listed each task that our team has coded and troubleshooted.

- Coupling Angular Error with Linear Error
- Bi-Directional Motion in Motion Controller
- Goal in Inflation Zones (Map Drifting)
- Collection of Data for PD Tuning & Motion Filter Weights
- Cubic Hermite Trajectory Generation

2.1. Coupling Angular Error with Linear Error

Before the implementation of coupling the angular error with linear error, the robot moves in a large arc towards its target. Consequently, the robot has the tendency of bumping into obstacles.

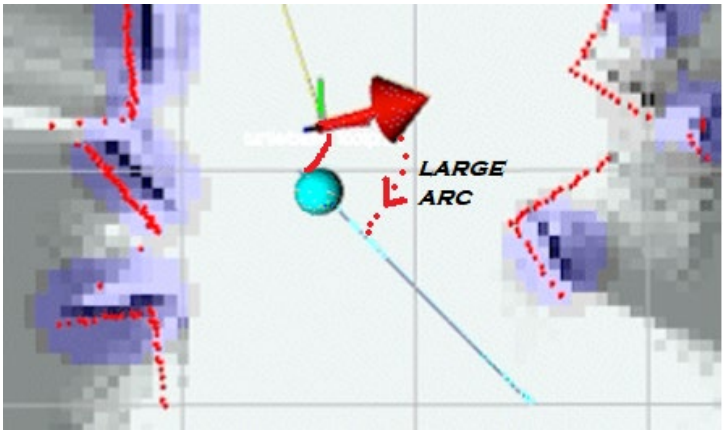

Figure 2: Robot moves in a large arc when target is behind

Initially, our team has came up with a function $u_{k,r} = f(P_{k,r} + I_{k,r} + D_{k,r}, \varepsilon_{k,\theta})$ that looks like Figure 2. The function f is supposed to output a value that is 1 when $\varepsilon_{k,\theta} = 0$, and output a value of 0 when $\varepsilon_{k,\theta}$ is some distance away from 0. In other words, the robot will prioritise rotating towards the target before moving linearly towards it. When the angular error $\varepsilon\_(k, \theta)$ is 0, the robot will move at its fullest speed as u_lin is multiplied by 1.

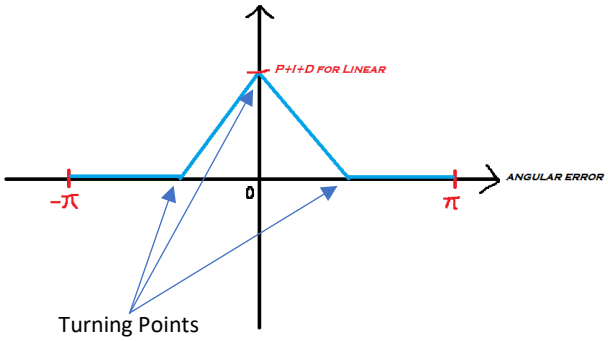

Figure 3: Coming up with the first function for the coupling

However, with this graph, we could see that the robot would not be able to accelerate/decelerate gradually at each turning point as shown in Figure 3. We decided to refine the function to make the turning points smoother. Using a graphical tool, we found that the cosine graph with degree of 4 can emulate this flexibility of producing various acceleration values with respect to the turning points. The graph was shown in Figure 4.

The function as shown below, where u_lin is overwritten

$$u\_lin = \begin{cases} u\_lin * cos(error\_ang)^4, & if\ abs(error\_ang) < \frac{\pi}{2} \\ 0, & otherwise \end{cases}$$

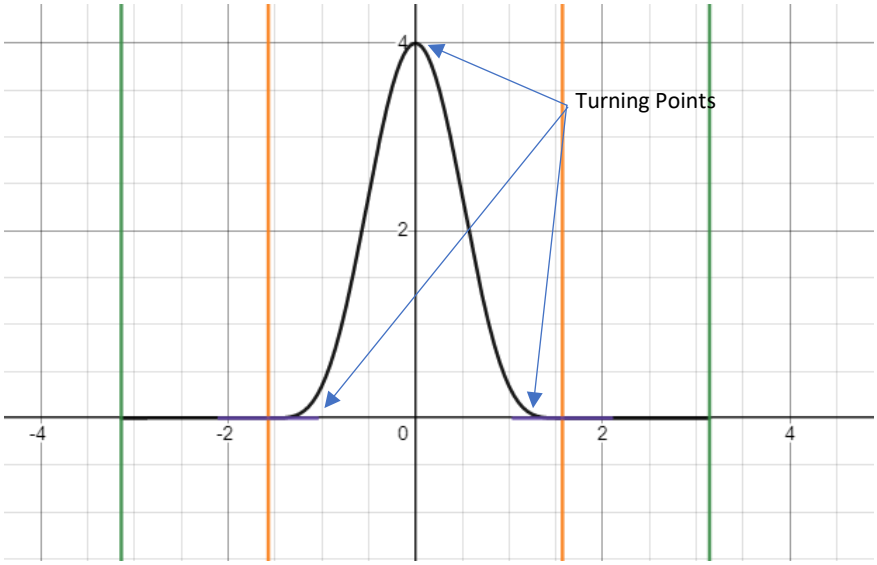

Figure 4: Smoother turning points compared to Figure 3

Initially, the code is as shown in Figure 5.

```
//  2.2 Coupling Angular Error with Linear Velocity
if (abs(error_ang)<(M_PI/2))
    u_lin = u_lin*pow(cos(error_ang),4);
else
    u_lin = 0;
```

Figure 5: Code for coupling function in Figure 4

After further limiting the angular error in the next section, Bi-Directional Motion, the code improved to just one line by removing the *if-else* statement. As shown below.

```
//  2.2 Coupling Angular Error with Linear Velocity
u_lin = u_lin*pow(cos(error_ang),4);
```

Figure 6: Improved code for coupling the errors

This is possible as the limit_angle in Figure X has limited the angular error to be within $-\frac{\pi}{2} < \varepsilon_{k,\theta} < \frac{\pi}{2}$. Anything outside the range won't be used by the robot, circled in red dashes as shown below.

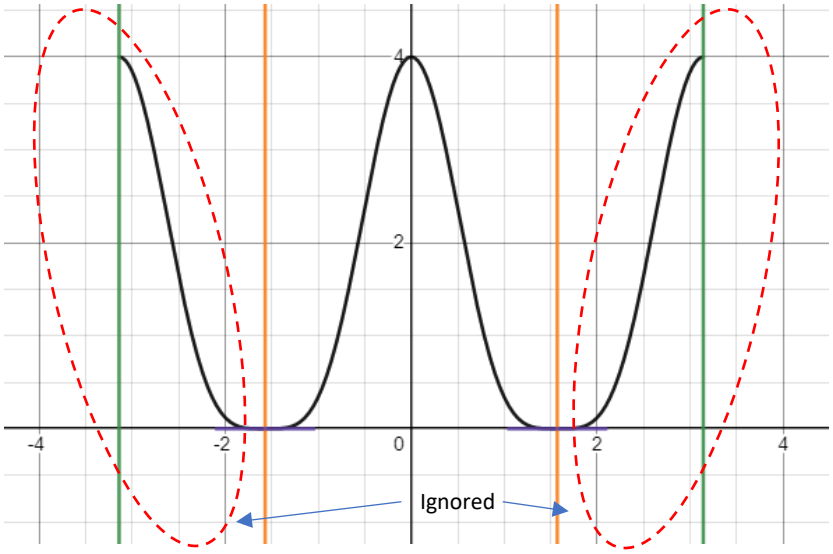

Figure 7: Limited coupling function

2.2. Bi-Directional Motion in Motion Controller

To reduce the need for rotating the robot, it is more convenient to spin towards $\varepsilon_{k,\theta} = \pi$ when $\varepsilon_{k,\theta} < -\frac{\pi}{2}$ or $\varepsilon_{k,\theta} > \frac{\pi}{2}$, and reverse towards its target.

```
error_lin = sqrt(pow((target.y - pos_rbt.y),2) + pow((target.x - pos_rbt.x),2));
error_ang = limit_angle(heading(pos_rbt,target)-ang_rbt);

if (error_ang >= M_PI_2 || error_ang <= -M_PI_2)
{
   error_ang = limit_angle(error_ang +M_PI);   //offset angular error by PI
   error_lin = -1 * error_lin;
}
```

Figure 8: Offsetting the angular error

In Figure X, π is added to the angular error when $\varepsilon_{k,\theta} < -\frac{\pi}{2}$ or $\varepsilon_{k,\theta} > \frac{\pi}{2}$. The linear error is also inverted in its direction, by multiplying with negative one.

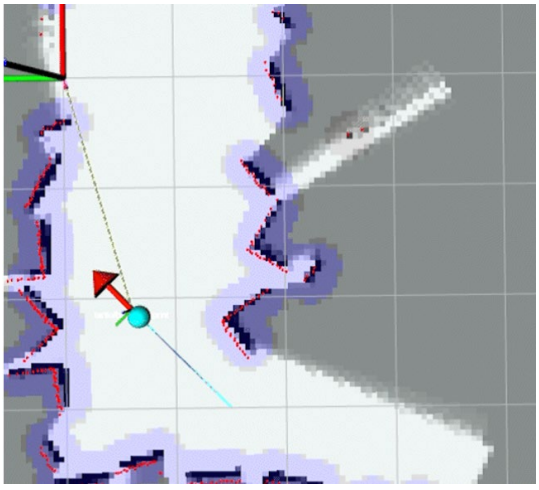

Figure 9: Robot is reversing to its target

Figure x shows the process that allows the robot to reverse to its target, instead of turning a huge angle to face and move linearly towards it.

2.3. Map Drifting (Goal in Inflation Zones)

When a goal is in inflation zones due to drifting, Dijkstra is used to find closest free space on the map. It will then re-plan the path for the robot to reach a goal that is the nearest from the inflation zone. Similar process is done to the robot position. When the robot enters inflation zone, it will re-publish a new position on the map that is at the nearest safe point. It will then re-plan the path accordingly.

For this feature, we verified it in simulation by inflating the inflation zone. Firstly, we define the starting robot position inside the inflation zone. Secondly, the goal is also set to be nearer to the boxes that is covered in inflation zone. The response in rqt_console shown below verifies the working function of Dijkstra algorithm.

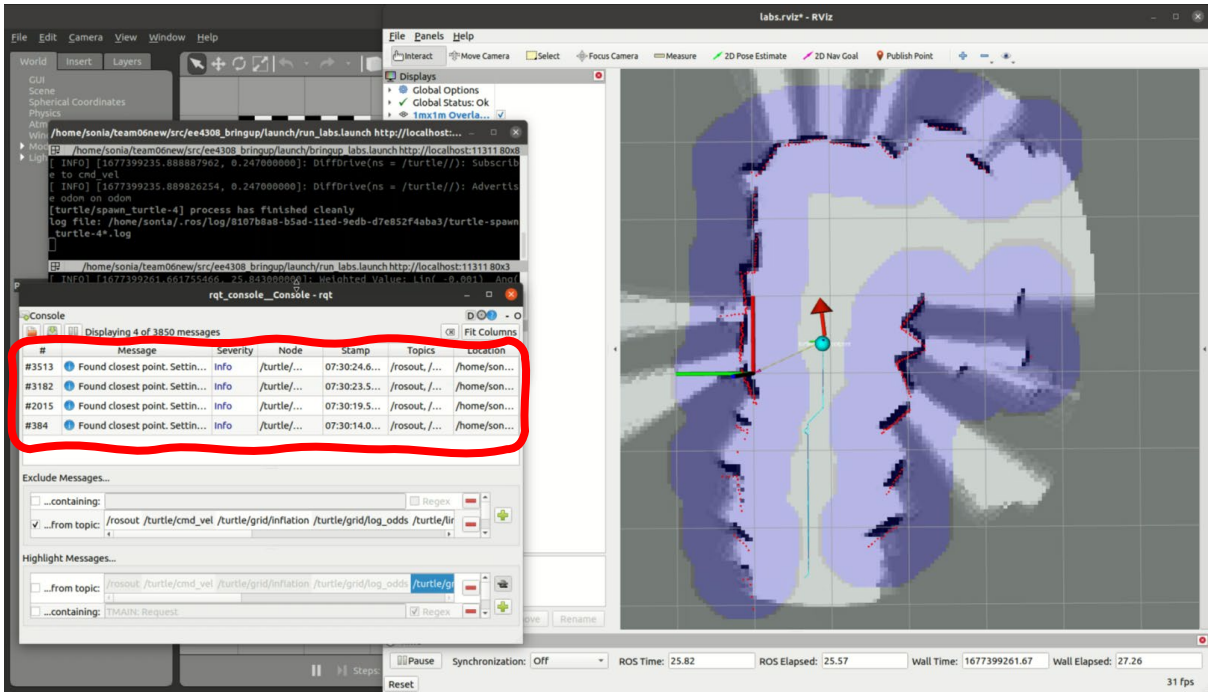

Figure 10:

The Dijkstra algorithm is activated when the robot position or goal target is inside the inflation zone. Dijkstra search function were declared in planner.cpp similar to **get**() function. As shown below, the Dijkstra function is modified from the **get**() function to feature "**is_closest_point**" Boolean to publish nearest safest point. **Is_closest_point** variable is updated by the if-else function that checks if the neighbouring costs lies in an accessible area.

```
224         if(is_closest_point){
225             ROS_INFO("Found closest point. Setting position...");
226 
227             path_idx_dijkstra.push_back(node->idx);
228 
229             break;
230         }
231
```

The switching happens in the main.cpp file as shown in the picture below. This code snippet shows the else argument section when the robot/goal position lies in an inaccessible cell. In this condition, **pos_rbt** and **pos_goal** are being assigned new value by calling the ***planner.dijkstra()*** function. After

which, the Boolean **move_out** is responsible to trigger a replan of path between the newly assigned position and goal.

```
// robot lies on inaccessible cell, or if goal lies on inaccessible cell
  if (!grid.get_cell(pos_rbt)){
      ROS_WARN(" TMAIN : Robot lies on inaccessible area. No path can be found");
      ROS_INFO(" TMAIN : Before Dijkstra[%.2f, %.2f]",pos_rbt.x, pos_rbt.y);
      Position prev_pos_rbt = pos_rbt;
      Position pos_rbt = planner.dijkstra(pos_rbt);
      // Position nearest_safe = planner.dijkstra(pos_rbt);
      ROS_INFO(" TMAIN : After Dijkstra [%.2f, %.2f]",pos_rbt.x, pos_rbt.y);
      // double head_to_safe_pos  = heading(pos_rbt, nearest_safe_pos);

      move_out = true;
```

Figure 11: Dijkstra algorithm activation mechanism

A Boolean variable – Controls the switch between A* and Dijkstra

```
28  void Planner::add_to_open(Node * node, bool swdjik)
29  {   // sort node into the open list
30      double node_f;
31
32      if (swdjik){
33          node_f = node->g;
34      }
35      else{
36          node_f = node->g + node->h;
37      }
```

Figure 12: Switching mechanism between A* and Dijkstra

2.4. Collection of Data

For the Turtlebot Burger to run smoothly to reach the target, it was necessary to tune PD gains and motion filter weights.

2.4.1. Tuning of Motion Filter Weights

The methods are similar to Lab 1.

To test for linear error, the robot was set to "**enable_move = true"** in .yaml file so that autonomous motion is de-activated. These are the values obtained under the testing conditions of moving forward at a distance 1.2m. The linear acceleration gradually increases to 1.2.

Linear Error			
run	weight_odom_v	weight_imu_v	error (metres)
1	0.5	0.5	0.02380
2	0.3	0.7	0.02690
3	0.6	0.4	0.00935
4	0.7	0.3	0.00929
5	0.8	0.2	0.00899
6	0.9	0.1	0.00817

Table 1: Linear error with different weights

To test for angular error, the robot was remained in the same setting as per testing for linear error to obtain "weight_odom_v". The testing conditions are unchanged, the robot must move forward to a distance of 1.2m, comes to a stop and spins in its spot. As its revolutions in one-direction, the angular acceleration gradually increases to its maximum value, -2.84.

Angular Error			
run	weight_odom_w	weight_imu_w	error (radians)
1	0.6	0.4	1.852
2	0.5	0.5	1.828
3	0.1	0.9	1.108
4	0.01	0.99	0.925
5	0.005	0.995	0.0723
6	0.004	0.996	0.0677

Table 2: Angular error with different weights

From the tele-operation settings, we can understand that the weight of the angular Odom is relatively low as compared to the analysis done in Lab 1. However, in Lab 1, we were testing motion filter under a simulation environment which negates potential factors like hardware components in Turtlebot that may affect the performance of the response of the weight Odom during autonomous motion. Factors such as wear and tear of motor gear tooth, IMU or LiDAR sensitivity and rusted wheel shaft.

Therefore, after the experiments, we proceed to test run these values in autonomous motion to observe the motion response and the mapping quality. Based on observations, we come to an understanding that autonomous motion we have chosen the values for the respective weights

- weight_odom_v: 0.9
- weight_imu_v: 0.1
- weight_odom_w: 0.004
- weight_imu_w: 0.996

2.4.2. Tuning of PD Gains

The method is similar to Lab 2, but we based on our analysis of the robot's motion as it navigates the maze to reach the 3 goals. These are our PID tuning values obtained with observing the effects of drift due to various hardware component factors. We then adjust accordingly with taking into respect of the "**target_dt" , "inflation_radius" ,** the maximum linear and angular acceleration values. We realised that setting a high value for maximum linear and angular acceleration may cause more instability in controlling the drifts. So we reduced our maximum linear and angular values to limit the drift and to also accommodate the constraints of the Turtlebot's hardware.

Linear Experiment				
run	Kp_lin	Kd_lin	Observation of motion [Test with PID ANG – Table 4]	Effects of Drift
1	1.0	0.1	Slow and steady, but map quality can be better	Minimal, Affects narrow corners
2	1.2	0.1	Moving steady, Map quality improved	Minimal, maintains map shape
3	0.9	0.1	Too slow to compensate for sudden reverse	Smeared map as it goes linear from turning.
4	4.4	0.3	Effects from Slippage is obvious	A lot
5	4.2	0.2	Slippage still present	Shifts in map
6	4.1	0.5	Slippage still present, slight effects of inertia as comes to stop	Shift in map

Table 3: Linear experiment with different PD values

Angular Experiment				
run	Kp_ang	Kd_ang	Observation of motion [Tested with PID LIN – Table 3]	Effects of Drift
1	1.0	0.05	Reduced the "smeared" shifts in the map caused by turning,	Minimal, maintains map shape
2	1.2	0.6	Corners scan slightly affected, but speed looks acceptable range for turns	Map has shift

3	1.2	0.2	Improved for corners scan, but Acceptable range of speed for turns	Distorts corners scan in map
4	1.7	0.6	Fast for turn settings, Unable to compensate the overshoot caused in motion change from linear to angular	Obvious in Map quality scan, Map is really distorted = location of pts shifted
5	0.1	0.05	Good balance of stability and control during turn, but slow	Minimal
6	1.0	0.05	Presence of control, but can be better	Slight drifts, but Map quality still alright = does not caused large distortion

Table 4: Angular experiment with different PD values

Therefore, after the experiments, we have chosen the values for the respective weights

- Kp_lin: 1.2
- Kd_lin: 0.1
- Kp_ang: 1.0
- Kd_and: 0.05

2.5. Cubic Hermite Trajectory Generation

To create a smoother path between the points on the global path. This is to prevent the robot from rotating at every point to face its new target. Taking hardware consideration of a raspberry pi Cubic hermite algorithm was adopted instead of Quintic hermite. This minimises limited computing capability of a raspberry pi to prevent it causing unwanted lag in other processes.

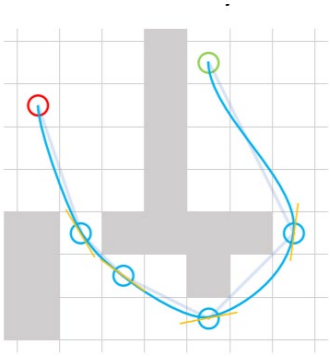

Figure 13: Cubic Hermite trajectory generation demonstration

Implementation of Cubic Hermite trajectory into turtlebot resulted in a smoother cornering to the robot's path. This trajectory algorithm is dependent on the speed of the robot between one turning point to another turning point. Hence, the default maximum linear speed of the turtlebot of 0.22 m/s will result in looping trajectory in the cornering points as shown below.

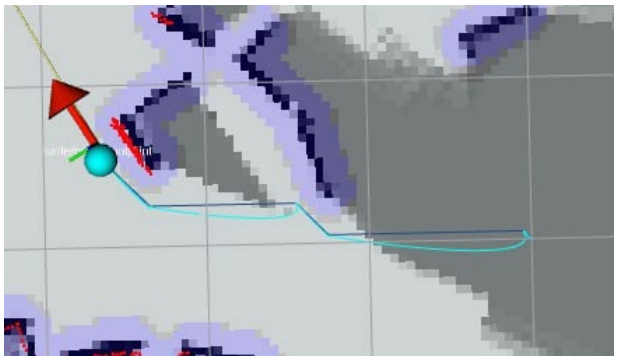

Figure 14: Smoother trajectory line demonstration

To resolve this, lowering its maximum linear speed to 0.16 m/s helped prevent this issue and created nice bends around corners.

3. Debug & Fix Any Errors

- Map drifting when the robot is turning a corner or accelerating when it is moving forward with no obstacles in front it
    - Given a high PID value, slippage is relatively present in the robot's motor gear due to the latency in change of velocity put in place in the Bi-directional control. So, it can cause the robot to drift away from its next actual target position. When it drifts, the lidar scan of the map could change by distorting the position of the free spaces and obstacles. Thus, making the robot unable to reach the proper target and navigating in the expected direction.
    - The first few runs when we are running our robot in the maze, we faced the issue whereby our robot is unable to reach the 3rd target. There were times where after a turn, it goes back into linear motion but hits part of the wall. Even as it is speeding,

the map drifts. We thought that adjusting the PID values only would eliminate the issue. However, we learnt that we should also adjust our maximum linear and angular values set in .yaml. The default values given are based on the actual specification by the supplier of the Turtlebot, so with consideration of external factors like electronic and hardware components, its limitations do play in affecting the expected performance of the robot's motion

    - When we have reduced our respective maximum value limits, there is a positive improvement shown in the robot's motion to map out a neat map layout of the maze.
    - when we are fixing the drift issue, we do not only investigate trajectory but also the values in .yaml. Adjusting the PID values means also adjusting our target_dt and the maximum limits of the Turtlebot.

- Trajectory errors
    - The turning point was unstable at high speed as it generates loops in cornering paths. This is solved by reducing the max speed of the turtlebot.
- Exit inflation zone
    - We place a few Boolean variables to switch between implementing Dijkstra path planner for inflation zones and A* path planner for the safe zones.

4. Conclusion

We intuitively understand that if given more time we could implement post-process improvements. The challenges we faced was in troubleshooting the above areas stated in Part 3 of our report, sometimes the issue can be hardware problems. So, we worked with our limitations and do our best to achieve the tasks required for us in this project.

All in all, this project 1 is a good exposure to the applications of robot mapping and autonomous motion. The debugging of fixes was worthwhile because we learnt a lot through mistakes. In the end, our project 1 felt a success though there could be improvements to be made for post-process, as we can successfully run our robot autonomously to hit the 3 targets of the maze.

5. Appendix A (code for main.cpp)

```
#include <ros/ros.h>

#include "grid.hpp"
#include "planner.hpp"
#include "trajectory.hpp"
#include <stdio.h>
#include <stdlib.h>
#include <cmath>
#include <errno.h>
#include <sensor_msgs/LaserScan.h>
#include <nav_msgs/Path.h>
#include <nav_msgs/OccupancyGrid.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/PointStamped.h>
#include <std_msgs/Float64.h>


bool move_out = false;
std::vector<float> ranges;
void cbScan(const sensor_msgs::LaserScan::ConstPtr &msg)
{
    ranges = msg->ranges; // creates a copy
}
Position pos_rbt(0, 0);
// Position nearest_safe_pos(0,0);
double ang_rbt = 10; // set to 10, because ang_rbt is between -pi and pi, and integer
for correct comparison while waiting for motion to load
void cbPose(const geometry_msgs::PoseStamped::ConstPtr &msg)
{
    auto &p = msg->pose.position;
    pos_rbt.x = p.x;
    pos_rbt.y = p.y;

    // euler yaw (ang_rbt) from quaternion <-- stolen from wikipedia
    auto &q = msg->pose.orientation; // reference is always faster than copying. but
changing it means changing the referenced object.
    double siny_cosp = 2 * (q.w * q.z + q.x * q.y);
    double cosy_cosp = 1 - 2 * (q.y * q.y + q.z * q.z);
    ang_rbt = atan2(siny_cosp, cosy_cosp);
}

double lin_vel;
void cblinvel(const std_msgs::Float64::ConstPtr &msg)
{
    lin_vel = msg->data; // creates a copy
}

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

    // Make sure motion and move can run (fail safe)
    nh.setParam("run", true); // turns off other nodes

    // Get ROS parameters
    bool verbose;
    if (!nh.param("verbose_main", verbose, true))
        ROS_WARN(" TMAIN : Param verbose_main not found, set to true");
    if (!nh.param("initial_x", pos_rbt.x, 0.))
        ROS_WARN(" TMAIN : Param initial_x not found, set to 0");
```

```
    if (!nh.param("initial_y", pos_rbt.y, 0.))
        ROS_WARN(" TMAIN : Param initial_x not found, set to 0");
    std::vector<Position> goals;
    std::string goal_str;
    if (nh.param("goals", goal_str, std::to_string(pos_rbt.x) + "," +
std::to_string(pos_rbt.y)))
    {
        char goal_str_tok[goal_str.length() + 1];
        strcpy(goal_str_tok, goal_str.c_str()); // to tokenise --> convert to c string
(char*) first
        char *tok = strtok(goal_str_tok, " ,");
        try
        {
            while (tok != nullptr)
            {
                goals.emplace_back(
                    strtod(tok, nullptr),
                    strtod(strtok(nullptr, " ,"), nullptr));
                ROS_INFO("GOALs %f, %f", goals.back().x, goals.back().y);
                tok = strtok(nullptr, " ,");
            }
        }
        catch (...)
        {
            ROS_ERROR(" TMAIN : Invalid Goals: %s", goal_str.c_str());
            ros::shutdown();
            return 1;
        }
    }
    else
        ROS_WARN(" TMAIN : Param goal not found, set to %s", goal_str.c_str());
    Position pos_min;
    if (!nh.param("min_x", pos_min.x, -10.))
        ROS_WARN(" TMAIN : Param min_x not found, set to -10");
    if (!nh.param("min_y", pos_min.y, -10.))
        ROS_WARN(" TMAIN : Param min_y not found, set to -10");
    Position pos_max;
    if (!nh.param("max_x", pos_max.x, 10.))
        ROS_WARN(" TMAIN : Param max_x not found, set to 10");
    if (!nh.param("max_y", pos_max.y, 10.))
        ROS_WARN(" TMAIN : Param max_y not found, set to 10");
    double close_enough;
    if (!nh.param("close_enough", close_enough, 0.1))
        ROS_WARN(" TMAIN : Param close_enough not found, set to 0.1");
    double target_dt;
    if (!nh.param("target_dt", target_dt, 0.04))
        ROS_WARN(" TMAIN : Param target_dt not found, set to 0.04");
    double average_speed;
    if (!nh.param("average_speed", average_speed, 0.2))
        ROS_WARN(" TMAIN : Param average_speed not found, set to 0.2");
    double cell_size;
    if (!nh.param("cell_size", cell_size, 0.2))
        ROS_WARN(" TMAIN : Param cell_size not found, set to 0.05");
    double inflation_radius;
    if (!nh.param("inflation_radius", inflation_radius, 0.2))
        ROS_WARN(" TMAIN : Param inflation_radius not found, set to 0.3");
    int log_odds_thresh;
    if (!nh.param("log_odds_thresh", log_odds_thresh, 10))
        ROS_WARN(" TMAIN : Param log_odds_thresh not found, set to 10");
    int log_odds_cap;
    if (!nh.param("log_odds_cap", log_odds_cap, 20))
        ROS_WARN(" TMAIN : Param log_odds_cap not found, set to 20");
```

```
    double main_iter_rate;
    if (!nh.param("main_iter_rate", main_iter_rate, 25.0))
        ROS_WARN(" TMAIN : Param main_iter_rate not found, set to 25");
    // print out the parameters
    ROS_INFO(" TMAIN : Goals[%s], Grid[%.2f,%.2f to %.2f,%.2f]  CloseEnuf:%f  TgtDt:%f
AvgSpd:%f  CellSize:%f  InfMskRad:%f  LOThrsh:%d  LOCap:%d",
             goal_str.c_str(), pos_min.x, pos_min.y, pos_max.x, pos_max.y,
close_enough, target_dt, average_speed, cell_size, inflation_radius, log_odds_thresh,
log_odds_cap);
    // subscribers
    ros::Subscriber sub_scan = nh.subscribe("scan", 1, &cbScan);
    ros::Subscriber sub_pose = nh.subscribe("pose", 1, &cbPose);
    ros::Subscriber sub_lin_vel = nh.subscribe("lin_vel", 1, &cblinvel);

    // Publishers
    ros::Publisher pub_path = nh.advertise<nav_msgs::Path>("path", 1, true);
    ros::Publisher pub_traj = nh.advertise<nav_msgs::Path>("trajectory", 1, true);
    ros::Publisher pub_target = nh.advertise<geometry_msgs::PointStamped>("target", 1,
true);
    ros::Publisher pub_grid_lo =
nh.advertise<nav_msgs::OccupancyGrid>("grid/log_odds", 1, true);
    ros::Publisher pub_grid_inf =
nh.advertise<nav_msgs::OccupancyGrid>("grid/inflation", 1, true);

    // Prepare published messages
    nav_msgs::OccupancyGrid msg_grid_lo, msg_grid_inf; // preparation below
    geometry_msgs::PointStamped msg_target;
    msg_target.header.frame_id = "world"; // for rviz
    nav_msgs::Path msg_path;
    msg_path.header.frame_id = "world"; // for rviz
    nav_msgs::Path msg_traj;
    msg_traj.header.frame_id = "world"; // for rviz

    // Setup the occupancy grid class
    Grid grid(pos_min, pos_max, cell_size, inflation_radius, log_odds_thresh,
log_odds_cap);

    // adjust msg_grid for rviz
    msg_grid_lo.data.resize(grid.size.i * grid.size.j);
    msg_grid_lo.header.frame_id = "world";
    msg_grid_lo.info.resolution = cell_size;
    msg_grid_lo.info.width = grid.size.j;
    msg_grid_lo.info.height = grid.size.i;
    msg_grid_lo.info.origin.position.x = -cell_size * 0.5 + pos_min.x;
    msg_grid_lo.info.origin.position.y = -cell_size * 0.5 + pos_min.y;
    // # flip the map bcos it is not displayed properly in rviz by rotating via
quaternion
    msg_grid_lo.info.origin.orientation.x = 1 / M_SQRT2;
    msg_grid_lo.info.origin.orientation.y = 1 / M_SQRT2;
    msg_grid_inf = msg_grid_lo; // copy over

    // Setup the planner class
    Planner planner(grid);

    // setup loop rates
    ros::Rate rate(main_iter_rate);

    // Other variables
    bool replan = true;
    std::vector<Position> path, post_process_path, trajectory;
    int g = 0;                      // goal num
    Position pos_goal = goals[g]; // to trigger the reach goal
```

```
    int t = 0;                    // target num
    Position pos_target;

    // wait for other nodes to load
    ROS_INFO(" TMAIN : Waiting for topics");
    while (ros::ok() && (ranges.empty() || ang_rbt == 10))
    {
        rate.sleep();
        ros::spinOnce(); // update the topics
    }

    ROS_INFO(" TMAIN : ===== BEGIN =====");

    while (ros::ok())
    {
        // update all topics
        ros::spinOnce();

        // update the occ grid
        grid.update(pos_rbt, ang_rbt, ranges);

        // publish the map
        grid.write_to_msg(msg_grid_lo, msg_grid_inf);
        pub_grid_lo.publish(msg_grid_lo);
        pub_grid_inf.publish(msg_grid_inf);

        if (dist_euc(pos_rbt, pos_goal) < close_enough)
        { // reached the goal, get new goal
            replan = true;
            if (++g >= goals.size())
            {
                if (verbose)
                    ROS_INFO(" TMAIN : Last goal reached");
                break;
            }
            // there are goals remaining
            pos_goal = goals[g];
        }
        else if (!is_safe_trajectory(trajectory, grid) || move_out == true)
        { // request a new path if path intersects inaccessible areas, or if there is
no path
            replan = true;
        }

        // always try to publish the next target so it does not get stuck waiting for
a new path.
        if (!trajectory.empty() && dist_euc(pos_rbt, pos_target) < close_enough ||
move_out == true)
        {
            if (--t < 0)
                t = 0; // in case the close enough for target triggers. indices cannot
be less than 0.

            pos_target = trajectory[t];

            if (verbose)
                ROS_INFO(" TMAIN : Get next target (%f,%f)", pos_target.x,
pos_target.y);

            // publish to target topic
            msg_target.point.x = pos_target.x;
            msg_target.point.y = pos_target.y;
```

```
            pub_target.publish(msg_target);
        }

        if (replan)
        {
            if (grid.get_cell(pos_rbt) && grid.get_cell(pos_goal) || move_out == true)
            {
                if (verbose)
                    ROS_INFO(" TMAIN : Request Path from [%.2f, %.2f] to Goal %d at
[%.2f,%.2f]",
                             pos_rbt.x, pos_rbt.y, g, pos_goal.x, pos_goal.y);
                // if the robot and goal are both on accessible cells of the grid
                path = planner.get(pos_rbt, pos_goal); // original path
                if (path.empty())
                { // path cannot be found
                    if (verbose)
                        ROS_WARN(" TMAIN : No path found between robot and goal");
                    // retry
                }
                else
                { // path found
                    // get turning points after processing path

                    // path.size() == 1 means start == goal. There is currently no
code to handle this case, which may lead to problems downstream (trajectory generation
and post process). Try and handle it.

                    if (verbose)
                        ROS_INFO(" TMAIN : Begin Post Process");
                    post_process_path = post_process(path, grid);

                    if (verbose)
                        ROS_INFO(" TMAIN : Begin trajectory generation over all
turning points");
                    // generate trajectory over all turning points
                    // doing the following manner results in the front of trajectory
being the goal, and the back being close to the rbt position
                    trajectory.clear();

                    for (int m = 1; m < post_process_path.size(); ++m)
                    {
                        Position &turn_pt_next = post_process_path[m - 1];  //position
of next turn
                        Position &turn_pt_cur = post_process_path[m];       //position
of current turn
                        //Position &turn_pt_next_next = post_process_path[m - 2];

                        //get initial velocity and final velocity
                        double i_vel_x = lin_vel * cos (ang_rbt);
                        double i_vel_y = lin_vel * sin (ang_rbt);

                        double f_angle = heading(turn_pt_cur, turn_pt_next); //heading
of next point from this point

                        //Added this
                        double Dx = (turn_pt_next.x - turn_pt_cur.x);
                        double Dy = (turn_pt_next.y - turn_pt_cur.y);
                        double time_taken = sqrt(Dx*Dx + Dy*Dy) / average_speed;
                        double arbitrary_magnitude =
sqrt(pow(Dx,2)+pow(Dy,2))/time_taken;

                        //double arbitrary_magnitude = 0.05;
```

```
                        //NO change here
                        double f_vel_x = arbitrary_magnitude * cos(f_angle); //vector
decoupling
                        double f_vel_y = arbitrary_magnitude * sin(f_angle); //vector
decoupling

                        // use line below for non-spline
                        // std::vector<Position> traj =
generate_trajectory(turn_pt_next, turn_pt_cur, average_speed, target_dt, grid);
                        // use line below for splaine
                        std::vector<Position> traj = generate_trajectory(turn_pt_next,
turn_pt_cur, average_speed, target_dt, grid, i_vel_x, i_vel_y, f_vel_x, f_vel_y);

                        for (Position &pos_tgt : traj)
                        {
                            trajectory.push_back(pos_tgt);
                        }
                    }

                    if (verbose)
                        ROS_INFO(" TMAIN : Trajectory generation complete");

                    // publish post processed path to path topic
                    msg_path.poses.clear();
                    for (Position &pos : post_process_path)
                    {
                        msg_path.poses.push_back(geometry_msgs::PoseStamped()); //
insert a posestamped initialised to all 0
                        msg_path.poses.back().pose.position.x = pos.x;
                        msg_path.poses.back().pose.position.y = pos.y;
                    }
                    pub_path.publish(msg_path);

                    // publish trajectroy to trajectory topic
                    msg_traj.poses.clear();
                    for (Position &pos : trajectory)
                    {
                        msg_traj.poses.push_back(geometry_msgs::PoseStamped()); //
insert a posestamped initialised to all 0
                        msg_traj.poses.back().pose.position.x = pos.x;
                        msg_traj.poses.back().pose.position.y = pos.y;
                    }
                    pub_traj.publish(msg_traj);

                    // get new target
                    t = trajectory.size() - 1; // last entry
                    // pick the more distant target so turtlebot does not stop
intermitently around very close targets when new path is generated
                    if (t > 15)
                        t -= 15; // this is the average_speed * 15 * target_dt away
                    pos_target = trajectory[t];

                    // publish to target topic
                    msg_target.point.x = pos_target.x;
                    msg_target.point.y = pos_target.y;
                    pub_target.publish(msg_target);

                    move_out = false;
                    replan = false;

                }
            }//end of if
```

```
            else
            { // robot lies on inaccessible cell, or if goal lies on inaccessible cell
                if (!grid.get_cell(pos_rbt)){
                    ROS_WARN(" TMAIN : Robot lies on inaccessible area. No path can be
                    found");
                    ROS_INFO(" TMAIN : Before Dijkstra[%.2f, %.2f]",pos_rbt.x,
                    pos_rbt.y);
                    Position prev_pos_rbt = pos_rbt;
                    Position pos_rbt = planner.dijkstra(pos_rbt);

                    ROS_INFO(" TMAIN : After Dijkstra [%.2f, %.2f]",pos_rbt.x,
                    pos_rbt.y);


                    move_out = true;


                    //exit inflation radius
                    if (pos_rbt.x - prev_pos_rbt.x > 0 ) {
                        pos_rbt.x += 0.001;
                    }
                    else if (pos_rbt.x - prev_pos_rbt.x < 0 ) {
                        pos_rbt.x -= 0.001;
                    }

                     if (pos_rbt.y - prev_pos_rbt.y > 0 ) {
                        pos_rbt.y += 0.001;
                    }
                    else if (pos_rbt.y - prev_pos_rbt.y < 0 ) {
                        pos_rbt.y -= 0.001;
                    }


                }

                if (!grid.get_cell(pos_goal)) {
                    ROS_WARN(" TMAIN : Goal lies on inaccessible area. No path can be
                    found");
                    pos_goal = planner.dijkstra(pos_goal);
                    ROS_INFO(" TMAIN : Request Path from [%.2f, %.2f] to Goal %d at
                    [%.2f,%.2f]", pos_rbt.x, pos_rbt.y, g, pos_goal.x, pos_goal.y);


                }

            }
        }

        // sleep for rest of iteration
        rate.sleep();
    }

    nh.setParam("run", false); // turns off other nodes
    ROS_INFO(" TMAIN : ===== END =====");
    return 0;
}
```

6. Appendix B (code for move.cpp)

```
#include <ros/ros.h>
#include <stdio.h>
#include <cmath>
#include <errno.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/PointStamped.h>
#include <geometry_msgs/Twist.h>
#include <std_msgs/Empty.h>
#include "std_msgs/Float64.h"
#include "common.hpp"
#include <fstream>

bool target_changed = false;
Position target;
void cbTarget(const geometry_msgs::PointStamped::ConstPtr &msg)
{
    target.x = msg->point.x;
    target.y = msg->point.y;
}

Position pos_rbt(0, 0);
double ang_rbt = 10; // set to 10, because ang_rbt is between -pi and pi, and integer
for correct comparison while waiting for motion to load
void cbPose(const geometry_msgs::PoseStamped::ConstPtr &msg)
{
    auto &p = msg->pose.position;
    pos_rbt.x = p.x;
    pos_rbt.y = p.y;

    // euler yaw (ang_rbt) from quaternion <-- stolen from wikipedia
    auto &q = msg->pose.orientation; // reference is always faster than copying. but
changing it means changing the referenced object.
    double siny_cosp = 2 * (q.w * q.z + q.x * q.y);
    double cosy_cosp = 1 - 2 * (q.y * q.y + q.z * q.z);
    ang_rbt = atan2(siny_cosp, cosy_cosp);
}

double get_pos_err(double xt,double yt,double xp,double yp) {
    double Dx = xt - xp;
    double Dy = yt - yp;

    return sqrt(Dx*Dx + Dy*Dy);
}

double get_ang_err(double xt,double yt,double xp,double yp, double heading) {
    double angular_error = atan2(yt-yp, xt-xp) - heading;

    angular_error = limit_angle(angular_error);

    return angular_error;
}

double motion_couple(double ang_err) {
    double x0 = 0;
    double x1 = 0.2;
    double y0 = 1;
    double y1 = 0;
    double x = abs(ang_err);
```

```
    double output = 0;
    // linear interpolate 0 to pi with 0 to 1
    // when ang_err = 0, output 1
    // when ang_err = some distance away, output 0

    output =((y1 - y0) / (x1 - x0) * (x - x0)) + y0;
    if (x > x1) {
        output = 0.0;
    }

    return abs(output);
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "turtle_move");
    ros::NodeHandle nh;
    // for recording data
    std::ofstream data_file;
    // data_file.open("/home/steve/a237r/data.txt");

    // Get ROS Parameters
    bool enable_move;
    if (!nh.param("enable_move", enable_move, true))
        ROS_WARN(" TMOVE : Param enable_move not found, set to true");
    bool verbose;
    if (!nh.param("verbose_move", verbose, false))
        ROS_WARN(" TMOVE : Param verbose_move not found, set to false");
    double Kp_lin;
    if (!nh.param("Kp_lin", Kp_lin, 1.0))
        ROS_WARN(" TMOVE : Param Kp_lin not found, set to 1.0");
    double Ki_lin;
    if (!nh.param("Ki_lin", Ki_lin, 0.0))
        ROS_WARN(" TMOVE : Param Ki_lin not found, set to 0");
    double Kd_lin;
    if (!nh.param("Kd_lin", Kd_lin, 0.0))
        ROS_WARN(" TMOVE : Param Kd_lin not found, set to 0");
    double max_lin_vel;
    if (!nh.param("max_lin_vel", max_lin_vel, 0.22))
        ROS_WARN(" TMOVE : Param max_lin_vel not found, set to 0.22");
    double max_lin_acc;
    if (!nh.param("max_lin_acc", max_lin_acc, 1.0))
        ROS_WARN(" TMOVE : Param max_lin_acc not found, set to 1");
    double Kp_ang;
    if (!nh.param("Kp_ang", Kp_ang, 1.0))
        ROS_WARN(" TMOVE : Param Kp_ang not found, set to 1.0");
    double Ki_ang;
    if (!nh.param("Ki_ang", Ki_ang, 0.0))
        ROS_WARN(" TMOVE : Param Ki_ang not found, set to 0");
    double Kd_ang;
    if (!nh.param("Kd_ang", Kd_ang, 0.0))
        ROS_WARN(" TMOVE : Param Kd_ang not found, set to 0");
    double max_ang_vel;
    if (!nh.param("max_ang_vel", max_ang_vel, 2.84))
        ROS_WARN(" TMOVE : Param max_ang_vel not found, set to 2.84");
    double max_ang_acc;
    if (!nh.param("max_ang_acc", max_ang_acc, 4.0))
        ROS_WARN(" TMOVE : Param max_ang_acc not found, set to 4");
    double move_iter_rate;
    if (!nh.param("move_iter_rate", move_iter_rate, 25.0))
        ROS_WARN(" TMOVE : Param move_iter_rate not found, set to 25");
```

```
    // Subscribers
    ros::Subscriber sub_target = nh.subscribe("target", 1, &cbTarget);
    ros::Subscriber sub_pose = nh.subscribe("pose", 1, &cbPose);

    // Publishers
    ros::Publisher pub_cmd = nh.advertise<geometry_msgs::Twist>("cmd_vel", 1, true);
    ros::Publisher pub_lin_error = nh.advertise<std_msgs::Float64>("move_lin_error",
1, true);
    ros::Publisher pub_rbt_pos = nh.advertise<geometry_msgs::Pose>("pos_rbt", 1,
true);

    // prepare published messages
    geometry_msgs::Twist msg_cmd; // all properties are initialised to zero.
    geometry_msgs::Pose rbt_pos;
    std_msgs::Float64 errorFloat;

    // Setup rate
    ros::Rate rate(move_iter_rate); // same as publishing rate of pose topic

    // wait for other nodes to load
    ROS_INFO(" TMOVE : Waiting for topics");
    while (ros::ok() && nh.param("run", true) && ang_rbt == 10) // not dependent on
main.cpp, but on motion.cpp
    {
        rate.sleep();
        ros::spinOnce(); //update the topics
    }

    // Setup variables
    double cmd_lin_vel = 0, cmd_ang_vel = 0;
    double dt;
    double prev_time = ros::Time::now().toSec();

    ////////////////// DECLARE VARIABLES HERE //////////////////
    double P_lin ,I_lin, D_lin,
    P_angle, I_angle, D_angle,
    forward_signal, angular_signal,
    prev_angular_signal,
    lin_acc, ang_acc,
    accum_error_lin, accum_error_ang,
    error_lin, ang_error,
    prev_error_lin, prev_ang_error = 0;

    double tenth_percentile = 0;
    double ninety_percentile = 0;
    double lin_max_overshoot = pos_rbt.x;
    double ang_max_overshoot = 0;

    bool captured10th = false;
    bool captured90th = false;
    bool experiment = false;
    bool linear_experiment = false;

    error_lin = get_pos_err(target.x, target.y, pos_rbt.x, pos_rbt.y);
    prev_error_lin = error_lin;

    ang_error = get_ang_err(target.x, target.y, pos_rbt.x, pos_rbt.y, ang_rbt);
    prev_ang_error = ang_error;

    ROS_INFO(" TMOVE : ===== BEGIN =====");

    // main loop
```

```
    if (enable_move)
    {
        while (ros::ok() && nh.param("run", true))
        {
            // update all topics
            ros::spinOnce();

            dt = ros::Time::now().toSec() - prev_time;
            if (dt == 0) // ros doesn't tick the time fast enough
                continue;
            prev_time += dt;

            // Time step control ---------------------------------------
            // note: monitor dt for its consistency in completing one cycle of
calculation
            // Question: is it better to set a constant time step than relying on dt ?
            // e.g. if (TIME_INC >= dt)

            ////////////////// MOTION CONTROLLER HERE //////////////////

            //Position error ------------------------------------------
            if (experiment) {
                error_lin = target.x - pos_rbt.x;
            }
            else {
                error_lin = get_pos_err(target.x, target.y, pos_rbt.x, pos_rbt.y);
            }

            //angular error -------------------------------------------
            //ang_error = get_ang_err(target.x, target.y, pos_rbt.x, pos_rbt.y,
ang_rbt);

            //omnidirection control
            ang_error = limit_angle(heading(pos_rbt,target) - ang_rbt);

            //if (abs(ang_error))

            // // //Coupling Angular Error with linear velocity

            // if (abs(ang_error)<= M_PI_2){
            //     if(abs(ang_error)<(M_PI/3)){ //(M_PI/3)?
            //         forward_signal = ( (forward_signal/2) * cos(3*ang_error) ) +
(forward_signal/2);
            //     }
            //     else{
            //         forward_signal = 0;
            //     }
            // }
            // else{
            //     if(abs(ang_error)>((2*M_PI)/3)){
            //         forward_signal = -1*( ((forward_signal/2)*cos( (3*ang_error) +
M_PI)) + (forward_signal/2) );
            //     }
            //     else{
            //         forward_signal = 0;
            //     }
            // }
            //error_lin*=pow(cos(ang_error),4);

            if (ang_error >= M_PI_2 || ang_error <= -M_PI_2) {
                ang_error = limit_angle(ang_error + M_PI);
                error_lin = -1 * error_lin; //forward_signal
```

```
}

//P control
P_lin = Kp_lin * error_lin;
//I control
accum_error_lin += error_lin;
I_lin = Ki_lin * accum_error_lin;
//D control
D_lin = Kd_lin * (error_lin - prev_error_lin) / dt;
prev_error_lin = error_lin;
forward_signal = P_lin + I_lin + D_lin;

forward_signal = forward_signal * pow(cos(ang_error),4);

lin_acc = (forward_signal - cmd_lin_vel) / dt;

if (experiment) {
    cmd_lin_vel = cmd_lin_vel + lin_acc * dt;
}
else {
    lin_acc = sat(lin_acc, max_lin_acc); //saturated forward signal
    cmd_lin_vel = sat(cmd_lin_vel + (lin_acc * dt), max_lin_vel);

// Angular error and linear velocity coupling -------------
//cmd_lin_vel = cmd_lin_vel * motion_couple(ang_error);
//in progress..
}


//P control
P_angle = Kp_ang * ang_error;
//I control
accum_error_ang += ang_error;
I_angle = Ki_ang * accum_error_ang;
//D control
D_angle = Kd_ang * (ang_error - prev_ang_error) / dt;
prev_ang_error = ang_error;

//PID TOTAL OF ANGULAR SIGNAL
angular_signal = P_angle + I_angle + D_angle;

//forward_signal = (P_lin + I_lin + D_lin)*pow(cos(ang_error),4);

ang_acc = (angular_signal - cmd_ang_vel) / dt;

if (experiment) {
    cmd_ang_vel = cmd_ang_vel + (ang_acc * dt);
}
else {
    ang_acc = sat(ang_acc, max_ang_acc); //saturated angular signal
    cmd_ang_vel = sat(cmd_ang_vel + (ang_acc * dt), max_ang_vel);
}



// publish speeds
if (linear_experiment == true && experiment == true) {
    msg_cmd.linear.x = cmd_lin_vel;
    msg_cmd.angular.z = 0;

    errorFloat.data = pos_rbt.x;
```

```
            pub_lin_error.publish(errorFloat);

            rbt_pos.position.x = pos_rbt.x;
            rbt_pos.position.y = pos_rbt.y;
            pub_rbt_pos.publish(rbt_pos);

            // calculate rise time
            if (pos_rbt.x > -2.99 && pos_rbt.x <-2.96 &&  !captured10th){
                tenth_percentile = ros::Time::now().toSec();
                captured10th = true;
            }

            if (pos_rbt.x > -2.84 && pos_rbt.x <-2.82 && !captured90th){
                ninety_percentile = ros::Time::now().toSec();
                captured90th = true;
            }

            //max_overshoot
            if (pos_rbt.x > target.x){
                if (pos_rbt.x - target.x > lin_max_overshoot){
                    lin_max_overshoot = pos_rbt.x - target.x;
                }
            }

        }
        else if (linear_experiment != true && experiment == true) {
            msg_cmd.linear.x = 0;
            msg_cmd.angular.z = cmd_ang_vel;

            rbt_pos.orientation.w = ang_rbt;
            pub_rbt_pos.publish(rbt_pos);

            // calculate rise time
            if ((ang_rbt > 0.3 && ang_rbt < 0.4) && !captured10th){
                tenth_percentile = ros::Time::now().toSec();
                captured10th = true;
            }

            if ((ang_rbt > 2.7 && ang_rbt < 2.9) && !captured90th){
                ninety_percentile = ros::Time::now().toSec();
                captured90th = true;
            }

            //max_overshoot
            if (ang_error < 0){  //atan2(target.x - pos_rbt.x, target.y -
pos_rbt.y);
                if (ang_error < ang_max_overshoot){ //heading of target
                    ang_max_overshoot = ang_error;
                }
            }

        }
        else {
            msg_cmd.linear.x = cmd_lin_vel;
            msg_cmd.angular.z = cmd_ang_vel;

            rbt_pos.position.x = pos_rbt.x;
            rbt_pos.position.y = pos_rbt.y;
            rbt_pos.orientation.w = ang_rbt;
            pub_rbt_pos.publish(rbt_pos);
        }
```

```
            pub_cmd.publish(msg_cmd);

            // verbose
            if (verbose)
            {
                if (linear_experiment) {
                    ROS_INFO("\t%6.3f\t%6.3f\t%7.3f\t%9.5f)\n", Kp_lin, Kd_lin,
ninety_percentile-tenth_percentile, lin_max_overshoot);
                }
                else {
                    ROS_INFO("\t%6.3f\t%6.3f\t%10.5f\t%9.5f)\n", Kp_ang, Kd_ang,
ninety_percentile-tenth_percentile,ang_max_overshoot);
                }
                // ROS_INFO(" TMOVE :  FV(%6.3f) AV(%6.3f) \n", cmd_lin_vel,
cmd_ang_vel);
                // ROS_INFO(" TIME :  TIME_INC (%6.3f) \n", dt);
                // ROS_INFO(" ang_rbt: %7.3f", ang_rbt);
                // ROS_INFO(" TIME :  TIME_INC (%6.3f) \n", dt);
                // ROS_INFO(" COUPLING: %7.3f \n", motion_couple(ang_error));
                // ROS_INFO(" ang_err: %7.3f\n", ang_error);
                // ROS_INFO("\t%f\t%f\t%f\t%f\t%f",
                // ros::Time::now().toSec(), error_lin, P_lin, accum_error_lin,
                // I_lin);
            }

            // write to file
            // if (linear_experiment) {
            //     data_file << ros::Time::now().toSec() << "\t" << error_lin
            //     << "\t" << P_lin << "\t" << accum_error_lin << "\t" <<
            //     I_lin << "\t" << D_lin << "\t" << cmd_lin_vel << "\t" << pos_rbt.x
<< "\t" << ninety_percentile << "\t" << tenth_percentile << "\t" << lin_max_overshoot
<< std::endl;
            // }
            // else {
            //     data_file << ros::Time::now().toSec() << "\t" << ang_error
            //     << "\t" << P_angle << "\t" << accum_error_ang << "\t" <<
            //     I_angle << "\t" << D_angle << "\t" << cmd_ang_vel << "\t" <<
ang_rbt << "\t" << ninety_percentile << "\t" << tenth_percentile << "\t" <<
ang_max_overshoot << std::endl;
            // }

            // wait for rate
            rate.sleep();
        }
    }

    // attempt to stop the motors (does not work if ros wants to shutdown)
    msg_cmd.linear.x = 0;
    msg_cmd.angular.z = 0;
    pub_cmd.publish(msg_cmd);

    ROS_INFO(" TMOVE : ===== END =====");

    // close file
    data_file.close();

    return 0;

}
```

7. Appendix C (code for planner.cpp)

```
#include "planner.hpp"
Planner::Node::Node()
    : g(0), h(0), visited(false), idx(-1, -1), parent(-1, -1)
    {}
Planner::Open::Open()
    : f(0), idx(-1, -1)
    {}
Planner::Open::Open(double f, Index idx)
    : f(f), idx(idx)
    {}
Planner::Planner(Grid & grid) // assumes the size of the grid is always the same
    : start(-1, -1), goal(-1, -1), grid(grid), nodes(grid.size.i * grid.size.j),
open_list()
    {
        // write the nodes' indices
        int k = 0;
        for (int i = 0; i < grid.size.i; ++i)
        {
            for (int j = 0; j < grid.size.j; ++j)
            {
                nodes[k].idx.i = i;
                nodes[k].idx.j = j;
                ++k;
            }
        }
    }


void Planner::add_to_open(Node * node, bool swdjik)
{   // sort node into the open list
    double node_f;

    if (swdjik){
        node_f = node->g;
    }
    else{
        node_f = node->g + node->h;
    }

    for (int n = 0; n < open_list.size(); ++n)
    {
        Open & open_node = open_list[n];
        if (open_node.f > node_f + 1e-5)
        {   // the current node in open is guaranteed to be more expensive than the
node to be inserted ==> insert at current location
            open_list.emplace(open_list.begin() + n, node_f, node->idx);

            // emplace is equivalent to the below but more efficient:
            // Open new_open_node = Open(node_f, node->idx);
            // open_list.insert(open_list.begin() + n, new_open_node);
            return;
        }
    }
    // at this point, either open_list is empty or node_f is more expensive that all
open nodes in the open list
    open_list.emplace_back(node_f, node->idx);
}
```

```
Planner::Node * Planner::poll_from_open()
{
    Index & idx = open_list.front().idx; //ref is faster than copy
    int k = grid.get_key(idx);
    Node * node = &(nodes[k]);

    open_list.pop_front();

    return node;
}

std::vector<Position> Planner::get(Position pos_start, Position pos_goal)
{
    std::vector<Index> path_idx = get(grid.pos2idx(pos_start),
grid.pos2idx(pos_goal));
    std::vector<Position> path;
    for (Index & idx : path_idx)
    {
        path.push_back(grid.idx2pos(idx));
    }
    return path;
}
std::vector<Index> Planner::get(Index idx_start, Index idx_goal)
{
    std::vector<Index> path_idx; // clear previous path

    // initialise data for all nodes
    for (Node & node : nodes)
    {
        node.h = dist_oct(node.idx, idx_goal);
        node.g = 1e5; // a reasonably large number. You can use infinity in clims as
well, but clims is not included
        node.visited = false;
    }

    // set start node g cost as zero
    int k = grid.get_key(idx_start);
    ROS_INFO("idx_start %d %d", idx_start.i, idx_start.j);
    ROS_INFO("idx_goal %d %d", idx_goal.i, idx_goal.j);
    Node * node = &(nodes[k]);
    node->g = 0;

    // add start node to openlist
    add_to_open(node, false);

    // main loop
    while (!open_list.empty())
    {
        // (1) poll node from open
        node = poll_from_open();

        // (2) check if node was visited, and mark it as visited
        if (node->visited)
        {   // if node was already visited ==> cheapest route already found, no point
expanding this anymore
            continue; // go back to start of while loop, after checking if open list
is empty
        }
        node->visited = true; // mark as visited, so the cheapest route to this node
is found
```

```
        // (3) return path if node is the goal
        if (node->idx.i == idx_goal.i && node->idx.j == idx_goal.j)
        {   // reached the goal, return the path
            ROS_INFO("reach goal");

            path_idx.push_back(node->idx);

            while (node->idx.i != idx_start.i || node->idx.j != idx_start.j)
            {   // while node is not start, keep finding the parent nodes and add to
open list
                k = grid.get_key(node->parent);
                node = &(nodes[k]); // node is now the parent

                path_idx.push_back(node->idx);
            }

            break;
        }

        // (4) check neighbors and add them if cheaper
        bool is_cardinal = true;
        for (int dir = 0; dir < 8; ++dir)
        {   // for each neighbor in the 8 directions

            // get their index
            Index & idx_nb_relative = NB_LUT[dir];
            Index idx_nb(
                node->idx.i + idx_nb_relative.i,
                node->idx.j + idx_nb_relative.j
            );

            // check if in map and accessible
            if (!grid.get_cell(idx_nb))
            {   // if not, move to next nb
                continue;
            }

            // get the cost if accessing from node as parent
            double g_nb = node->g;
            if (is_cardinal)
                g_nb += 1;
            else
                g_nb += M_SQRT2;
            // the above if else can be condensed using ternary statements: g_nb +=
is_cardinal ? 1 : M_SQRT2;

            // compare the cost to any previous costs. If cheaper, mark the node as
the parent
            int nb_k = grid.get_key(idx_nb);
            Node & nb_node = nodes[nb_k]; // use reference so changing nb_node changes
nodes[k]
            if (nb_node.g > g_nb + 1e-5)
            {   // previous cost was more expensive, rewrite with current
                nb_node.g = g_nb;
                nb_node.parent = node->idx;

                // add to open
                add_to_open(&nb_node,false); // & a reference means getting the
pointer (address) to the reference's object.
            }

            // toggle is_cardinal
```

```
            is_cardinal = !is_cardinal;
        }
    }

    // clear open list
    open_list.clear();
    return path_idx; // is empty if open list is empty
}

Position Planner::dijkstra(Position rbt_position){
    std::vector<Index> path_idx_dijkstra = dijkstra(grid.pos2idx(rbt_position));
    Position safe_pt;

    for(Index & idx : path_idx_dijkstra){
        if(grid.get_cell(grid.idx2pos(idx))){   //if grid is accessible
            safe_pt = grid.idx2pos(idx);        //assign nearest available point on
grid
            break;
        }
    }

    return safe_pt;
}
std::vector<Index> Planner::dijkstra(Index idx_current){
    std::vector<Index> path_idx_dijkstra;
    bool is_closest_point = false;
    // initialise data for all nodes
    for (Node & node : nodes)
    {
        node.g = 1e7; // a reasonably large number. You can use infinity in clims as
well, but clims is not included
        node.visited = false;
    }

    // set start node g cost as zero
    int k = grid.get_key(idx_current);
    ROS_INFO("idx_start %d %d", idx_current.i, idx_current.j);
    Node * node = &(nodes[k]);
    node->g = 0;

    // add start node to openlist
    add_to_open(node,true);

    // main loop
    while (!open_list.empty())
    {
        // (1) poll node from open
        node = poll_from_open();

        // (2) check if node was visited, and mark it as visited
        if (node->visited)
        {   // if node was already visited ==> cheapest route already found, no point
expanding this anymore
            continue; // go back to start of while loop, after checking if open list
is empty
        }
        node->visited = true; // mark as visited, so the cheapest route to this node
is found

        if(is_closest_point){
            ROS_INFO("Found closest point. Setting position...");
```

```
            path_idx_dijkstra.push_back(node->idx);

            break;
        }

        // (4) check neighbors and add them if cheaper
        bool is_cardinal = true;
        for (int dir = 0; dir < 8; ++dir)
        {   // for each neighbor in the 8 directions

            // get their index
            Index & idx_nb_relative = NB_LUT[dir];
            Index idx_nb(
                node->idx.i + idx_nb_relative.i,
                node->idx.j + idx_nb_relative.j
            );

            // check if in map and accessible
            if (!grid.get_cell(idx_nb))
            {   // if not, move to next nb
                continue;
            }

            // set is_closest_point true if neighbors are accessible and direction is
at final itiration
            if(grid.get_cell(idx_nb) && dir == 7){
                is_closest_point = true;
            }


            // get the cost if accessing from node as parent
            double g_nb = node->g;
            if (is_cardinal)
                g_nb += 1;
            else
                g_nb += M_SQRT2;
            // the above if else can be condensed using ternary statements: g_nb +=
is_cardinal ? 1 : M_SQRT2;

            // compare the cost to any previous costs. If cheaper, mark the node as
the parent
            int nb_k = grid.get_key(idx_nb);
            Node & nb_node = nodes[nb_k]; // use reference so changing nb_node changes
nodes[k]
            // if (nb_node.g > g_nb + 1e-5)
            // {   // previous cost was more expensive, rewrite with current
                nb_node.g = g_nb;
                nb_node.parent = node->idx;

                // add to open
                add_to_open(&nb_node,true); // & a reference means getting the pointer
(address) to the reference's object.
            // }

            // toggle is_cardinal
            is_cardinal = !is_cardinal;
        }
    }

    // clear open list
    open_list.clear();
    return path_idx_dijkstra; // is empty if open list is empty
```

```
}
```

8. Appendix D (code for trajectory.cpp)

```
#include "trajectory.hpp"


std::vector<Position> post_process(std::vector<Position> path, Grid &grid) // returns
the turning points
{
    // (1) obtain turning points
    if (path.size() <= 2)
    { // path contains 0 to elements. Nothing to process
        return path;
    }

    // add path[0] (goal) to turning_points
    std::vector<Position> turning_points = {path.front()};
    // add intermediate turning points
    for (int n = 2; n < path.size(); ++n)
    {
        Position &pos_next = path[n];
        Position &pos_cur = path[n - 1];
        Position &pos_prev = path[n - 2];

        double Dx_next = pos_next.x - pos_cur.x;
        double Dy_next = pos_next.y - pos_cur.y;
        double Dx_prev = pos_cur.x - pos_prev.x;
        double Dy_prev = pos_cur.y - pos_prev.y;

        // use 2D cross product to check if there is a turn around pos_cur
        if (abs(Dx_next * Dy_prev - Dy_next * Dx_prev) > 1e-5) // if vectors are
parallel, cross product = 0
        {   // cross product is small enough ==> straight
            turning_points.push_back(pos_cur);
        }
    }
    // add path[path.size()-1] (start) to turning_points
    turning_points.push_back(path.back());

    std::vector<Position> post_process_path;

    // (2) make it more any-angle
    // done by students

    post_process_path = turning_points; // remove this line if (2) is done
    return post_process_path;
}

std::vector<Position> generate_trajectory(Position pos_begin, Position pos_end, double
average_speed, double target_dt, Grid & grid, double iv_x, double iv_y, double fv_x,
double fv_y)
{
    // (1) estimate total duration
    double Dx = pos_end.x - pos_begin.x;
    double Dy = pos_end.y - pos_begin.y;
    double duration = sqrt(Dx*Dx + Dy*Dy) / average_speed;

    double a0_x = pos_begin.x;
    double a1_x = iv_x;
    double a2_x = ( (-3* pos_begin.x)/(duration * duration) ) + ( (-2*iv_x
)/(duration) ) + ((3*pos_end.x)/(duration * duration)) + ( (-1*fv_x) / duration);
```

```
    double a3_x = (( 2 * pos_begin.x)/(duration * duration * duration) ) +
(iv_x/(duration*duration)) + ( (-2* pos_end.x)/(duration*duration*duration) ) + (fv_x
/ (duration * duration));

    double a0_y = pos_begin.y;
    double a1_y = iv_y;
    double a2_y = ( (-3* pos_begin.y)/(duration * duration) ) + ((-2* iv_y) / duration
) + ((3* pos_end.y)/ (duration * duration) ) + ((-1*fv_y)/ duration);
    double a3_y = ( (2* pos_begin.y)/(duration * duration * duration) ) +
(iv_y/(duration*duration)) + ( (-2*pos_end.y)/(duration*duration*duration) ) + (fv_y /
(duration * duration));


    // (2) generate cubic / quintic trajectory
    // done by students
    std::vector<Position> trajectory = {pos_begin};
    for (double time = target_dt; time < duration; time += target_dt)
    {
        //interpolate position
        pos_begin.x = a0_x + (Dx * time / duration); //current pos
        // xi_dot = a1_x;
        pos_end.x = a0_x + (a1_x * time) + (a2_x * pow(time,2)) + (a3_x *
(pow(time,3)) );
        // xf_dot = a1_x + (a2_x * 2 * time) + (a3_x * 3 * (time * time));

        pos_begin.y = a0_y + (Dy * time / duration);
        // xi_dot = a1_x;
        pos_end.y = a0_y + (a1_y * time) + (a2_y * pow(time,2)) + (a3_y *
(pow(time,3)));
        // xf_dot = a1_y + (a2_y * 2 * time) + (a3_y * 3 * (time * time));

        trajectory.emplace_back(pos_end.x, pos_end.y);
    }

    // OR (2) generate targets for each target_dt
    // std::vector<Position> trajectory = {pos_begin};
    // for (double time = target_dt; time < duration; time += target_dt)
    // {
    //     trajectory.emplace_back(
    //         pos_begin.x + Dx*time / duration,
    //         pos_begin.y + Dy*time / duration
    //     );
    // }

    return trajectory;
}

// void calculate_acceleration (double initial_pos, double initial_speed, double
final_pos, double final_speed, double time) {
//     double a0 = initial_pos;
//     double a1 = initial_speed;
//     double a2 = (-3/(time * time) * initial_pos) - (2 / time * initial_speed) + (3
/ (time * time) * final_pos) - (final_speed / time);
//     double a3 = (2/(time * time * time) * initial_pos) +
(initial_speed/(time*time)) - (2/(time*time*time) * final_pos) + (final_speed / (time
* time));

//     initial_pos = a0;
//     initial_speed = a1;
//     final_pos = a0 + (a1 * time) + (a2 * (time * time)) + (a3 * (time * time *
time));
//     final_speed = a1 + (a2 * 2 * time) + (a3 * 3 * (time * time));
```

```
//     return initial_pos, initial_speed, final_pos, final_speed;
// }

bool is_safe_trajectory(std::vector<Position> trajectory, Grid & grid)
{   // returns true if the entire path is accessible; false otherwise
    if (trajectory.size() == 0)
    {   // no path
        return false;
    }
    else if (trajectory.size() == 1)
    {   // goal == start
        return grid.get_cell(trajectory.front()); // depends on the only cell in the
path
    }

    // if there are more than one turning points. Trajectory must be fine enough.
    for (int n=1; n<trajectory.size(); ++n)
    {
        if (!grid.get_cell(trajectory[n]))
            return false;
        // Use this if the trajectory points are not fine enough (distance >
cell_size)
        // Index idx_src = grid.pos2idx(trajectory[n-1]);
        // Index idx_tgt = grid.pos2idx(trajectory[n]);

        // grid.los.reset(idx_src, idx_tgt); // interpolate a straight line between
points; can do away with los if points are fine enough.
        // Index idx = idx_src;
        // while (idx.i != idx_tgt.i || idx.j != idx_tgt.j)
        // {
        //     if (!grid.get_cell(idx))
        //     {
        //         return false;
        //     }
        //     idx = grid.los.next();
        // }
        // comment out till here
    }
    return true;
}
```