Contest I Report

GROUP 9

Gary Leung, 1002177155

Litos (Hanze) Li, 1002526493

Justin (Zhaocong) Yuan, 1002352777

Jojo (Yizhi) Zhou, 1003002396

February 11, 2020

Contents

1 Problem Definition

1.1 Objective

The main objective for contest one is to simultaneously localizing and creating the map for a given environment with a TurtleBot autonomously. To be more concrete, the team is responsible for developing an algorithm for robot exploration that can autonomously navigate through a maze, identify environment/obstacles within and create a map within a time limit.

1.2 Requirements and Constraints

- The TurtleBot must autonomously navigate and map a contest environment consisting of 4.87x4.87 m2 room with static objects. The output of this task will be a file with the mapped environmental data.
- The TurtleBot will have a time limit of 8 minutes to perform the mapping and navigation. Once this time limit is reached, all activity must stop and the mapping results must be saved and provided to the Instructor/TAs.
- The TurtleBot must perform the task autonomously without any human intervention.
- The TurtleBot must use sensory feedback to facilitate the navigation of the environment. The robot cannot simply follow a fixed sequence of actions without sensor help.
- The TurtleBot should follow a max speed limitation of 0.25m/s when navigating the environment. This limit is increased to 0.1m/s when the robot is near any obstacles such as walls.
- The exploration algorithm should be designed to be robust to unknown environments with static obstacles.

2 Strategies

Our main design objective is to maximize the robustness and efficiency at which the TurtleBot explores the environment and to minimize unknown behaviors that might cause failure or noisy map production. Therefore, we designed our main design strategy based on simple wall-following and built upon that with other strategies such as bumper response, probabilistic random exploration, etc. The following Table 1 briefly summarizes the strategy functionalities along with their pros and cons for our design.

Table 1: Strategies Summary

Strategy 1	**Wall-following**
Func	Follow the walls and explore the environment
Pros	Simple yet robust for exploring the environment and less prone to failing
Cons	Robot could get stuck at a portion of the map without exploring the whole environment based on the geometry and size of the map
Strategy 2	**State Transition**
Func	A Bernoulli process to transit between different modes of exploration, with a tuned probabilistic parameter
Pros	Make up the disadvantage of simple wall-following algorithm that could trap the robot at a portion of the map
Cons	Probabilistic model so no guarantee of behavior in unknown environment at later phases of exploration
Strategy 3	**Rotate for Direction**
Func	Rotate 360 degrees and look for optimal direction to move forward
Pros	Increase map completeness by sensing more environment and choose optimal paths
Cons	Time consuming and the optimal direction does not guarantee the discovery of a new frontier for mapping.
Strategy 4	**Bumper**
Func	Step back and re-explore for free path when bumper activated
Pros	Avoided unexpected collision into walls or obstacles
Cons	Could not detect collisions from upper part of the robot

3 Robot Design and Implementation

Figure 1 shows the general architecture for the TurtleBot design. Sensor streams provide environment information to support all decision making and control. Low-level control contains all primitives to support robot movements, and high-level control ensures the navigation and mapping process is both robust and efficient.

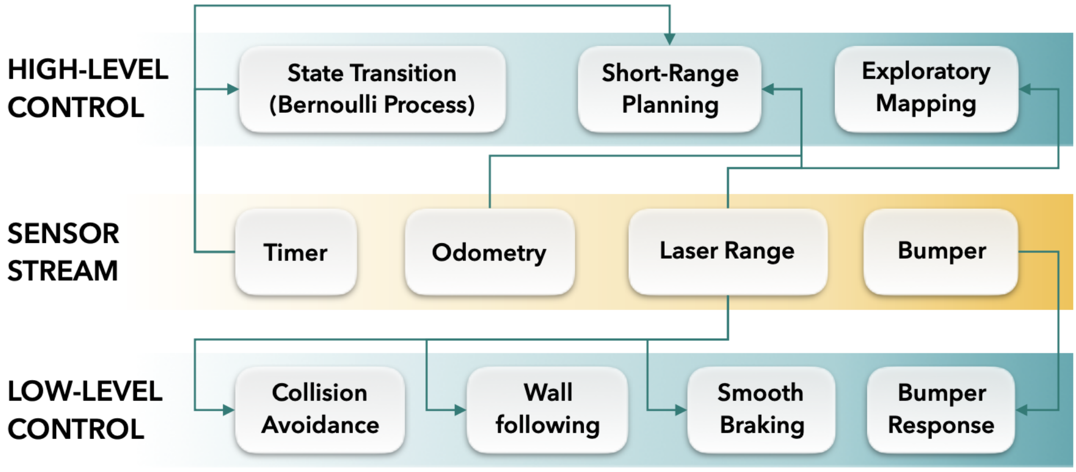

Figure 1: Overview of robot architecture

3.1 Sensory Design

Multiple sensor units are available for use in the TurtleBot, including 3 cliff sensors, 2 wheel drop sensors, 3 front bumpers, a gyroscope, an odometer, an RGB camera, a depth sensor, and a microphone array. Among the available sensory units, three groups of sensors were utilized in the navigation algorithm of the TurtleBot, including the depth sensors, the odometer, and the set of bumpers.

3.1.1 Bumper Sensors

Three bumpers are located at the front of the base of the TurtleBot, with each of the bumpers covering 60 degrees of the frontal area. The corresponding bumper will be triggered when the TurtleBot run into obstacles and provide information for the robot exploration algorithm to respond. In the case of our algorithm, triggering the bumpers will cause the robot to stop and attempt to free itself from colliding from an obstacle. This process is gone into more details in the Bumper section in 3.2.3 Collision Avoidance.

3.1.2 Laser Scan Sensors

The laser sensor gives a view of 60 degrees in front and is used for guiding the robot to perform exploration and construct the map. Since the laser sensor is not accurate when moving with high speed, especially when the TurtleBot is spinning, we limit the speed of the robot to be below 0.25 m/s while exploring in open area and 0.1 m/s while near obstacles and walls. From the laser messages, we primarily look at 3 values: the minimum left laser range, the minimum right laser range, and the overall minimum laser range. We obtain these values by performing a sweeping search across values obtained within the ranges array obtained from the sensor, while ignoring values that are below the minimal laser range. We split the laser range equally in half and denote them as the left field and the right field. By using these three values, we can maintain a good balance between obstacles on both our right and left sides during exploration.

3.1.3 Odometry callback

From our odometry callback we obtain our estimated x and y position, as well as our current yaw with respect to our starting position. Using these values, we can estimate the distance that the TurtleBot has moved, both in long and short timeframes, which can aid in our program's decision making process. In addition to this, the yaw value from the odometry callback is crucial when we wish to make precise turns, such as when we position ourselves for frontier exploration.

3.2 Controller Design

Following our design principle, the controller design used in contest 1 is mainly developed based on the simple wall-following algorithm with proportional and derivative control. Since wall following would only allow the robot to explore the connected area where walls/boundaries exist

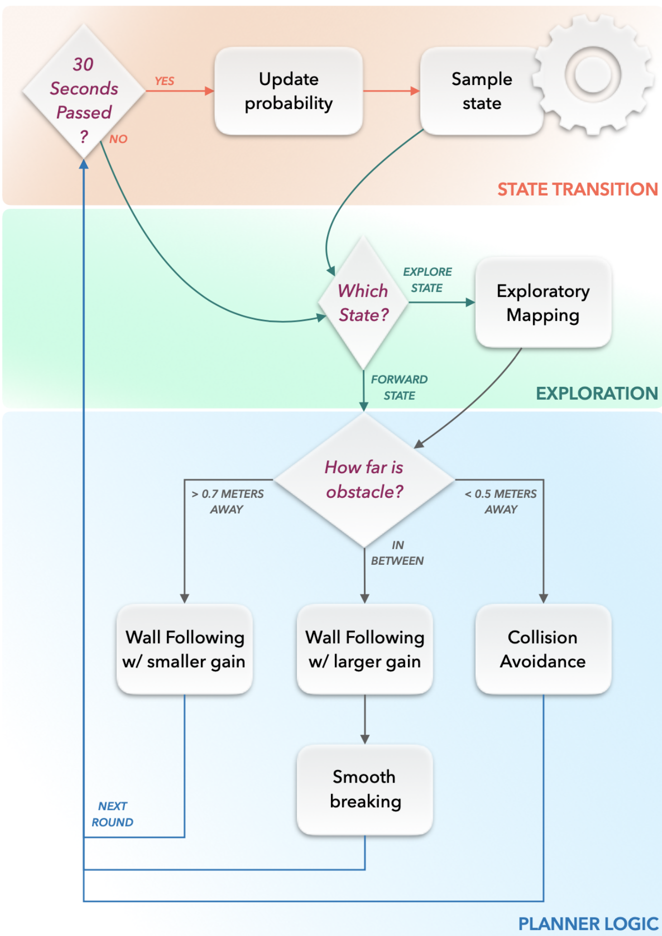

Figure 2: Flowchart of controller design

and exists many problems such as obstacle avoidance and exploration into the center of the environment being explored, we add several features on top of it to achieve a simple yet efficient algorithm for exploring the environment in 8 minutes. Figure 2 shows a flowchart of the overall controller logic. In the following, we will describe each component in details.

3.2.1 High-level Controller Design

Our high-level controller combines deterministic and random exploration models. The deterministic model always explores the furthest path, i.e. goes wherever direction with widest open frontier according to the senors, while the random model simply chooses a random path from available options.

Deterministic exploration algorithm

To guide navigation within the map, we model the robot as a Bernoulli process $\{X_t\}_{t\in T}$ consisting of two states $\mathcal{S} = \{F, E\}$: forward and explore. The process is non-stationary and the probability of falling in forward state decays linearly in time. Each sampled state is executed for certain time interval Δt, we use 30 seconds in practice.

$$X_t \sim Ber(p_t), \;\; \mathcal{P}(X_t = F) = p_t, \;\; p_t \propto \frac{T-t}{T}$$

- `Forward State`. The robot simply goes forward and turns only if necessary to avoid collision with walls. This is done by a reactive PID that maintains a margin with the closest wall.
- `Explore State`. The robot stops every 2.0m traversed, rotates around to (re-)map surrounding regions and then decides on the best direction to explore. To compensate for the overhead in exploratory mapping, we want to follow a smooth trajectory and maximally avoid collision. This is achieved by an aggressive PID controller that forces robot to stay away from any walls.

Random exploration algorithm

For random exploration, we use an reverse ϵ-greedy style annealing schedule to control the amount of stochasticity. In conventional ϵ-greedy exploration, the controller policy is given by

$$\pi(a|s) = \frac{\epsilon}{|\mathcal{A}|} + (1-\epsilon)\mathbf{1}(a, \operatorname*{argmax}_{a'} U(s, a'))$$

where s is the current available information (bumper signal, laser distances, partially built map and controller state), a is the current action being considered, $\mathcal{A}$ is the action space, $\mathbf{1}(a, \operatorname{argmax}_{a'} U(s, a'))$ will favor the action that maximizes some utility function U and ϵ is the exploration rate that decays in time.

In our design, we prioritize guided exploration in early phase leveraging sensor data, and gradually switch to uniform exploration to cover potential blind spots due to deterministic exploration. It is achieved by increasing the exploration rate according to a linear schedule in time $\epsilon \propto \frac{t}{T}$, where t and T are current and total run time.

3.2.2 Low-level Controller Design

In the following we give a more detailed explanation on how each part of the algorithm is constructed including wall-following, usage of bumper, random and deterministic exploration model.

Collision Avoidance

In most cases, the robot should explore the environment without hitting any wall or obstacles. This is done by rotating towards the opposite direction of any potential obstacles via checking the depth sensor. More specifically, several functions were designed in order for the TurtleBot turn while following the wall:

- `stayAwayFromWalls`: Adjustment Function - the robot ensures that a safe distance is maintained from both left and right side obstacles. This is done by rotating away from any objects that the laser scanner have detected as too close.
- `stayCentered`: Adjustment Function - the robot ensures that it moves in a centered manner between both left and right side obstacles. Unlike stayAwayFromWalls, stayCentered differs in that it actively controls the robot to stay at a balanced distance between left and right side obstacles, instead of simply ensuring that the obstacles are outside a minimal safe distance.
- `stayChill`: Adjustment Function - the robot moves with decaying speed the closer it is to an obstacle. This results in more accurate mapping, as well as a larger response time to maneuver away from obstacles when needed.
- `rotate2explore`: Rotate function - the robot will rotate clockwise or counter clockwise to look for "safe" direction to move where obstacles in that path is far away enough from the TurtleBot. This function was called when front sensor reading is too low, meaning there's an obstacles in front of the robot.
- `chooseDirection`: Correction function - the robot will rotate 360 degrees first. Then it will rotate to the direction that has the most space. The robot should choose direction on its left or right side prior to the front side. Also, the robot will not turn back. This function was called periodically (every threshold distance was travelled by the TurtleBot) or whenever the front bumper was activated.

However, during experiments and tests, TurtleBot may still occasionally hit obstacles or stuck at certain location due to sensor reading errors and laptop on top interfering the environment.

Therefore, bumper was leveraged during exploration and once one of the three bumpers is activated, the robot will act correspondingly to avoid stuck in.

Bumper Design

Bumper was used primarily for accidental bumping into walls or obstacles. In case of the bumper is activated, we first read which one of the three bumpers is activated and then act as follows:

- Middle bumper activated: to avoid the obstacle or wall in front of the TurtleBot, we stop and then step back for a set amount of distance, before using the chooseDirection() function described in the previous section to choose optimal path to explore. In practice, we use 0.18 meters as our set distance.
- Right/Left bumper activated: as the robot is too close to a right/left obstacles or walls, we execute a maneuver to avoid this obstacle. The robot will first move back a set amount of distance (the same as set above) and then the robot will turn clockwise/counter-clockwise away from obstacles. We then move forward the same amount of distance before angling back again towards our initial direction of motion. This allows us to avoid an obstacle by effectively shifting our position perpendicularly away from the obstacle through our four point maneuver.

3.3 Map Generation

The map was produced using the ROS built-in gmapping package. Figure 3 shows an example map rendered from ROS gmapping after running o simulation.

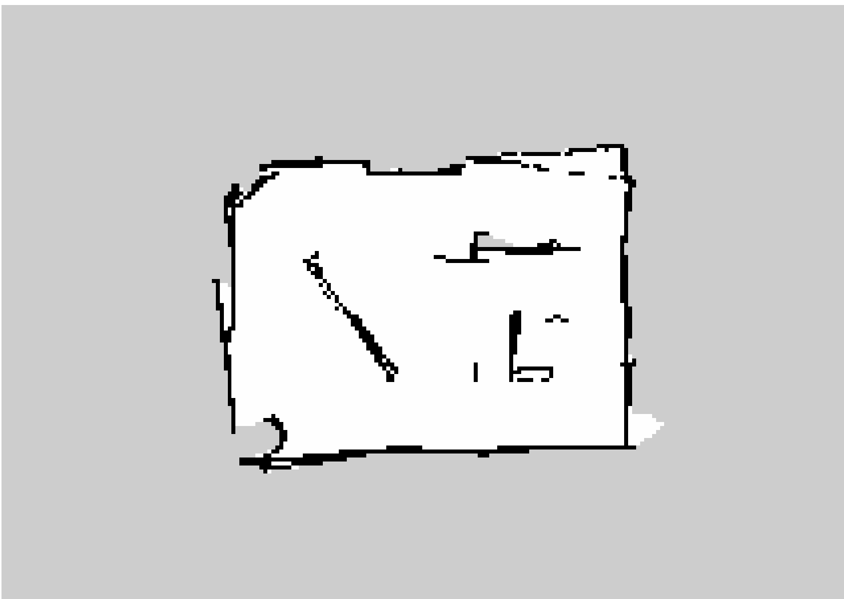

Figure 3: Example map from ROS gmapping

4 Future Recommendations

One concern we still have with our current design is that the TurtleBot may not be able to explore the whole environment, as we are leveraging a probabilistic approach to explore in the later phase. However, there lacks a systematical approach for exploring the entire environment efficiently without the need to go around or re-explore. A potential solution to this problem is to leverage the Occupancy Grid Map given in the ROS packages to keep track of places visited before and try to explore non-visited areas while possible. However, there are also drawbacks since generating and processing the Occupancy Grid takes a non-negligible amount of time and the localization of the TurtleBot in the map may not be accurate especially when errors accumulate as time goes. Therefore, there is a trade-off between efficiency and completeness for this method given limited resources.

Another issue with our current design is the noise caused by rapid linear and angular motion of the TurtleBot during exploration. Due to rapid initiation and/or termination of movement, disturbances such as vibration of the robot will be included as data. These disturbances often cause various artifacts within our generated maps, such as out-of-boundary areas, or inaccurate shape detection (particularly with curved objects). Theoretically, a simple proportional type of control can solve this problem and smoothen out the motion. We may also optimize our control further through tuning parameter operations. In addition, outside of the scope of the contest, there are many approaches that can help with the accuracy of the generated noise. We could potentially use extra sensors not present within the TurtleBot (such as the laptop camera) to help with odometry and mapping. In addition to this, it is possible to implement post-processing measures clean the map output.

Appendix A - Team Contributions

Table 2 is the contribution of each teammates in Contest1.

Table 2: Contest1 Team Contribution.

	Gary	Litos	Justin	Jojo
Overall Design	×	×	×	×
Exploration Algorithm			×	×
Sensor design	×	×		
Obstacle Avoidance and Bumpers	×	×		×
Pipeline	×		×	
Simulation and Testing	×	×	×	×
Parameter Tuning			×	×
Report	×	×	×	×

Appendix B - C++ ROS Code (2 marks)

Main ROS code are in two files, the main file named ”contest1.cpp”, and ”planner.cpp” which contains the exploration functions.

```
// #include "mie443_contest1/include/planners.h"
#include "planners.h"

/* Public Functions */

void motionPlanner::startup()
{
    ROS_INFO("Performing Startup");
    chooseDirection();
}

void motionPlanner::step()
{
    ROS_INFO("Stepping");
```

```
    ros::spinOnce();
    plannerMain();
}


/* Private Functions */

//////////////////////////////////////////////////////////////////////////////////////////
// Planning functions ////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////

void motionPlanner::plannerMain()
{
    float angular, linear;
    // Offset Calculation
    int left_index = laserSize - 1 - laserOffset, right_index = laserOffset;
    int minLeftLaserIndex = left_index - ((laserSize - 1) / 2);
    int minRightLaserIndex = ((laserSize - 1) / 2) - right_index;

    // Reevaluate the state every certain duration
    if (time_passed - time_last_update >= time_step) setState();

    ROS_INFO("Current State: %d, LeftRange: %f, RightRange: %f", state, minLeftLaserDist, minRightLaserDist);

    // Choose direction if in exploration state
    if (state == EXPLORE){
        if (dist(prevX, prevY, posX, posY) > explore_per_dist)
        {
            prevX = posX;
            prevY = posY;
            chooseDirection();
        }
    }

    // Check and then start moving if everything is fine
    ros::spinOnce();
    eStop.block();
    checkBumpers();
```

```


    // ----------------------------------------------------------------------------------------
    // ---------- START OF MAIN CONTROL LOGIC --------------------------------------------
    // ----------------------------------------------------------------------------------------

    if (minLaserDist > obstacleDist+obstacleDist_zone)
    {
        if (state = EXPLORE)
        {
            angular = stayCentered(minLeftLaserDist, minRightLaserDist, minLeftLaserIndex, minRightLaserIndex, k_p_small, 0.);
        }
        else {
            angular = 0.;
        }

        linear = linear_max;
        angular = stayAwayFromWalls(minLeftLaserDist, minRightLaserDist, angular);
    }

    // When the front sensor reading is too low
    else if (minLaserDist < obstacleDist-obstacleDist_zone)
    {
        // Determine which side has more space
        if (minRightLaserDist < minLeftLaserDist) {
            rotate2explore();
        }
        else {
            rotate2explore(CW);
        }
    }

    // When you are in the safe but not thaaaat safe zone, so turn faster and also be ready to slow down
    else
    {
```

```
        angular = stayCentered(minLeftLaserDist, minRightLaserDist, 0., 0., k_p_big, 0.);
        linear = stayChill(minLaserDist);
    }

    // ------------------------------------------------------------------------------------
    // ---------- END OF MAIN CONTROL LOGIC -----------------------------------------------
    // ------------------------------------------------------------------------------------


    // write the defined speed to the robot
    ROS_INFO("Main Publishing Velocity");
    publishVelocity(angular, linear, true /* SpinOnce */);
}


/////////////////////////////////////////////////////////////////////////////////////////
// Bumper Chdck /////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////

void motionPlanner::checkBumpers() {
    if (bumper[kobuki_msgs::BumperEvent::LEFT] == 1 ||
        bumper[kobuki_msgs::BumperEvent::CENTER] == 1 ||
        bumper[kobuki_msgs::BumperEvent::RIGHT] == 1)
    {
        float startX, startY;
        bool right = bumper[kobuki_msgs::BumperEvent::RIGHT] == 1;
        bool center = bumper[kobuki_msgs::BumperEvent::CENTER] == 1;

        // Maneuver backwards
        startX = posX; startY = posY;
        while (dist(startX, startY, posX, posY) < bumperPullbackDist)
        {
           publishVelocity(0 /* angular */, -0.1 /* linear */, true /* spinOnce */);
        }

```

```
        // If it's the front bumper, redirect the robot
        if (center)
        {
            chooseDirection();
        }
        // If side, do a "__/" kind of reruoting
        else
        {
            // Adjust angle away from obstacle
            if (right)
                rotate2angle(20);
            else
                rotate2angle(20, CW);

            // Maneuver forwards
            startX = posX;
            startY = posY;
            while (dist(startX, startY, posX, posY) < 0.15)
            {
                publishVelocity(0 /* angular */, 0.1 /* linear */, true /* spinOnce */);
            }

            // Adjust angle back to original direction
            if (right)
                rotate2angle(20, CW);
            else
                rotate2angle(20);
        }
    }
}

geometry_msgs::Twist motionPlanner::threeRegion() {
    /**
     * Controls the robot to follow the wall. Uses three regions control.
     * @param  {minLaserDist} float : the minimum value in msg->ranges from our laser scan - distance to closest wall.
     */
    geometry_msgs::Twist output;

    float closeThreshold = 0.5;
```

```
    float currentDistFromWall = minLaserDist;
    float controlYaw = 0;
    float forwardSpeed = 0.1;

    // If we turn outwards if we are too close to wall and we turn inwards if we are too far from wall.
    if (currentDistFromWall < closeThreshold)
    {
        controlYaw = 2;
        forwardSpeed = 0;
    }
    else if (currentDistFromWall > 2)
    {
        controlYaw = -1;
    }
    publishVelocity(controlYaw, forwardSpeed);
}

////////////////////////////////////////////////////////////////////////////////////////
// Rotation //////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////

bool motionPlanner::inRange(int bin, const vector<double> & binRange, bool front /* = false */){
    /**
     * Check if the bin is in the desired zone
     */
    if (front){
        return bin > binRange[0] || bin < binRange[1];
    }
    else {
        return bin > binRange[0] && bin < binRange[1];
    }
}

void motionPlanner::rotate2angle(float angle, bool CCW /* = true */) {
    /**
     * Rotate the robot to disred angle
     * @param  {degree} float : degree in degrees
```

```
     * @param  {CCW} bool : default rotation is CCW == turn left, set to
    false if need to turn right
     */
    // define velocity
    double angular = angular_max;
    if (!CCW)
        angular = angular * -1;

    // constraints
    ros::spinOnce();
    currYaw = yaw;
    double rad = DEG2RAD(angle); // TODO: maybe move it to before passing
    to rotate

    // rotate until desired
    while (abs(yaw - currYaw) < rad)
    {
        publishVelocity(angular, 0.0, true /* SpinOnce */);
    }
}

void motionPlanner::rotate2explore(bool CCW /* = true */) {
    /**
     * Rotate the robot until it is heading to a further wall/object
     * @param  {CCW} bool : default rotation is CCW == turn left, set to
    false if need to turn right
     */
    double angular = angular_max;
    if (!CCW)
        angular = angular * -1;

    // Stop turning (ready to go forward linearly) if there's something
    far away enough
    while (minLaserDist < exploreDist || minLeftLaserDist <
    exploreDist_lr || minRightLaserDist < exploreDist_lr)
    {
        publishVelocity(angular, 0.0, true /* SpinOnce */);
    }
}

void motionPlanner::rotate2bin(int bin) {
```

```
    /**
     * Command center of which rotate to perform
     * @param  {int} bin : bin index
     */
    if (bin < exploreAngle_bins / 2)
    {
        rotate2angle(bin * exploreAngle_size);
    }
    else
    {
        rotate2angle((exploreAngle_bins - bin) * exploreAngle_size, CW);
    }
}

void motionPlanner::chooseDirection() {
    /**
     * Rotate and choose direction to explore
     */
    // Define variables to store maximum value
    float maxDist_front = 0., maxDist_side = 0.; // default is 0 zo that if things go weird, it just moves forward
    int maxDist_front_idx = 0, maxDist_side_idx = 0;

    // Explore the front/left/right zones (no back zone!)
    for (int bin = 0; bin < exploreAngle_bins; bin++)
    {
        ros::spinOnce();

        // TODO: logic is fine, but might need to change the code appearance
        if (inRange(bin, exploreZone_front, FRONT))
        {
            if (minLaserDist > maxDist_front)
            {
                maxDist_front = minLaserDist;
                maxDist_front_idx = bin;
            }
        }
        else if (inRange(bin, exploreZone_left) || inRange(bin, exploreZone_right))
        {
```

```
            if (minLaserDist > maxDist_side)
            {
                maxDist_side = minLaserDist;
                maxDist_side_idx = bin;
            }
        }
        rotate2angle(exploreAngle_size);
    }

    if (maxDist_front < obstacleDist && maxDist_side < obstacleDist_side)
    { // Turn back around if its a dead end
        rotate2angle(turnBack);
    }
    else if (maxDist_side > exploreDist_side)
    { // Prefers to turn to the side in this state
        rotate2bin(maxDist_side_idx);
    }
    else
    { // If neither, go forward
        rotate2bin(maxDist_front_idx);
    }
}

//////////////////////////////////////////////////////////////////////////////////////////
// Adjustment functions ////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////
float motionPlanner::stayAwayFromWalls(float leftDist, float rightDist, float default_angular) {
    if (leftDist < 0.5) {
        return -angular_max;
    }
    else if (rightDist < 0.5) {
        return angular_max;
    }
    else {
        return default_angular;
    }
}

```

```
float motionPlanner::stayCentered(float leftDist, float rightDist, int leftIndex, int rightIndex, float k_p, float default_angular){
    float curr_diff_lr = leftDist - rightDist;
    int curr_diff_index = leftIndex - rightIndex;

    float angular = default_angular;

    // Distance difference can't be too large
    if (curr_diff_lr > allowed_laser_diff_lr) {
        angular = k_p * curr_diff_lr / leftDist;
    }
    else if (-curr_diff_lr > allowed_laser_diff_lr) {
        angular = k_p * curr_diff_lr / rightDist;
    }

    // Index difference can't be too large either (otherwise orientation is skewed)
    if (curr_diff_index > allowed_laser_diff_index)
        angular = -angular_max;
    else if (-curr_diff_index > allowed_laser_diff_index)
        angular = angular_max;

    return angular;
}

float motionPlanner:: stayChill(float frontDist, float default_linear){
        float k_p_linear = 0.8; // even i
        float dangerousDist = frontDist - obstacleDist; // if < 0, dangerous

        if (dangerousDist <= -0.5 * obstacleDist_zone) {
            k_p_linear = 0.3;
        }
        else if (dangerousDist <= obstacleDist_zone) {
            k_p_linear = 0.5;
        }
        else if (dangerousDist <= 0) {
            k_p_linear = 0.65;
        }
        else if (dangerousDist <= 0.5 * obstacleDist_zone) {
            k_p_linear = 0.7;
```

```
        }

        return k_p_linear * linear_max;
}


////////////////////////////////////////////////////////////////////////////////////////
// Helper functions ////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////

float motionPlanner::dist(float startX, float startY, float endX, float endY)
{
    return sqrt(pow(endX - startX, 2) + pow(endY - startY, 2));
}

void motionPlanner::publishVelocity(float angular, float linear, bool spinOnce /*= false*/)
{
    ROS_INFO("Publishing - Linear: %f, Angular: %f", linear, angular);
    geometry_msgs::Twist vel;
    vel.angular.z = angular;
    vel.linear.x = linear;
    vel_pub.publish(vel);

    if (spinOnce)
    {
        ros::spinOnce();
    }
}

////////////////////////////////////////////////////////////////////////////////////////
// State decision //////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////

void motionPlanner::setState() {
    /**
     * Choose a state depending on the time passed, all variables are global
```

```
     */
    std::random_device device;
    std::mt19937 gen(device());

    ros::spinOnce();
    time_passed =

    std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()
    - time_start).count();
    random_prob = time_passed / time_total;

    std::bernoulli_distribution randomOrNot(random_prob);
    goRandom = randomOrNot(gen);
    if (goRandom) {
        state = EXPLORE;
    } else {
        state = FORWARD;
    }
    time_last_update = time_passed;

    ROS_INFO("%f seconds, state: %d, random_output: %d", (float)
    time_last_update, state, goRandom);
    return;
}

//////////////////////////////////////////////////////////////////////////////////////////
// Callback functions
    ///////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////

void motionPlanner::bumperCallback(const
    kobuki_msgs::BumperEvent::ConstPtr& msg)
{
    // Access using bumper[kobuki_msgs::BumperEvent::{}] LEFT, CENTER, or
    RIGHT
    bumper[msg->bumper] = msg->state;
}

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

```
    nLasers = (msg->angle_max - msg->angle_min) / msg->angle_increment;
    desiredNLasers = DEG2RAD(desiredAngle)/msg->angle_increment;
    laserOffset = desiredAngle * M_PI / (180 * msg->angle_increment);
    laserScanTime = msg->scan_time;
    ROS_INFO("Size of laser scan array: %i, size of offset: %i, angle_max: %f, angle_min: %f, range_max: %f, range_min: %f, angle_increment: %f, scan_time: %f",
        nLasers, desiredNLasers, msg->angle_max, msg->angle_min, msg->range_max, msg->range_min, msg->angle_increment, msg->scan_time);

    minLaserDist = msg->range_max;
    minLeftLaserDist = msg->range_max;
    minRightLaserDist = msg->range_max;
    for (uint32_t laser_idx = nLasers / 2 - desiredNLasers; laser_idx < nLasers / 2 + desiredNLasers; ++laser_idx)
    {
        if (msg->range_max > msg->ranges[laser_idx] > 0)
        {
                minLaserDist = std::min(minLaserDist, msg->ranges[laser_idx]);
        }
    }
    for (uint32_t laser_idx = 0; laser_idx < nLasers/2; ++laser_idx)
    {
        if (msg->range_max > msg->ranges[laser_idx] > 0)
        {
                minRightLaserDist = std::min(minRightLaserDist, msg->ranges[laser_idx]);
        }
        if (msg->range_max > msg->ranges[nLasers-laser_idx-1] > 0)
        {
                minLeftLaserDist = std::min(minLeftLaserDist, msg->ranges[nLasers-laser_idx-1]);
        }
    }
    ROS_INFO("MinLaserDist: %f, minLeftLaserDist: %f, minRightLaserDist: %f", minLaserDist, minLeftLaserDist, minRightLaserDist);
}

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

```
    posX = msg->pose.pose.position.x;
    posY = msg->pose.pose.position.y;
    yaw = tf::getYaw(msg->pose.pose.orientation);
    ROS_INFO("Position: (%f, %f) Orientation: %f rad or %f degrees.",
   posX, posY, yaw, RAD2DEG(yaw));
}
```