Range-Only 3D Simultaneous Localization and Mapping using Ultra-Wideband Sensors

Zheng Yao
zheng.yao@mail.utoronto.ca
#1234567890

Abhishek Goudar
abhishek.goudar@mail.utoronto.ca
#1234567890

Fall 2019

Contents

1 Problem Definition and Objectives

Autonomous exploration and mapping of previously unknown environments is essential for countless robotics applications ranging from performing search and rescue [1] to aiding household chores [2]. The purpose of this contest is to identify and develop the essential components of a robotics system that can autonomously explore and map a previously unknown environment.

1.1 Problem Definition

The high level objective of the contest is the following:

> "Given an enclosed environment containing several static obstacles, design a robotics systems that can autonomously explore this environment with the purpose of producing a complete and accurate map.

Each team will be given a total of two trials that each can take up to 8 minutes in time to navigate the environment. In this 8 minutes, the robot is expected to use efficiently explore the given environment and create a map of it using the ROS Gmapping algorithms. This map is also expected to contain the location of key landmarks that will also be present in the environment. The layout of the environment will not be revealed prior to the competition, although it will have an area of $3.66 \times 4.90m^2$ and will contain multiple static obstacles.

A view of a sample environment is given in Figure 1

Figure 1: View of a potential environment layout.[3]

To make the task more concrete, we proceed with a detailed list of sub-objectives, metrics to measure success and constraints.

1.2 Detailed Objectives, Metrics and Constraints

A successful entry in the contest is expected to achieve the following two goals, measured by the following metrics (Table 1).

Objective	Description	Success Metric
High Quality Map	The generated map should contain data for all the parts of the enclosed environment.	Percentage of the environment correctly mapped, as compared to the ground truth map.
Landmarks	The key landmarks in the environment should be clearly visible in the generated map.	Percentage of the key landmarks detected. The landmarks will be in a cylindrical shape of radius approximately 40cm, as can be seen in Figure 1

Table 1: Objectives and success metrics.

1.3 Constraints

There are constraints regarding the the time limits, speed of the robot, the amount of human intervention, the ROS packages available and robot hardware (Table 2). There are also soft constraints on the quality of the map generated. For example, the map should contain the least possible amount of distortion introduced by odometry drift.

Constraint	Description
Time Limit	Each trial must be completed in 8 minutes or less.
Speed	In order to guarantee the safety and encourage higher mapping quality, the robot cannot move faster than 0.25m/s away from obstacles and 0.1m/s when close to an obstacles.
ROS Packages	The ROS packages that are available to each team is also limited. Therefore, each team is supposed to use the Gmapping package to perform simultaneous localization and mapping.
Human Intervention	The robot must run fully autonomously after it is launched.
Robot Hardware	Each team is required to use the sensors and actuators already present on TurtleBot 2 [4] and use a Lenovo 11e Laptop to control the robot.

Table 2: Objectives and success metrics.

1.4 Contest Settings

We now detail the contests settings, with an emphasis on the robot that will be used and the mapping task.

1.4.1 Robot Details

All teams will be given a TurtleBot [4] to use in the competition. The base of TurtleBot is equipped with numerous sensors including three bumpers (to detect collisions), odometry sensors

(to measure the movement of left and right wheels), a gyroscope (to keep track of rotation), three cliff sensors (to detect cliffs and alitute changes), two wheel drop sensors (to detect when one or both of the wheels lose contact of the ground), three docking IR receivers (for docking) and a Microsoft Kinect sensor [5] (for monocular RGB imaging and depth sensing). A Lenovo 11e Laptop, which is located on the top platform of the TurtleBot, is used to control to behavior of the robot.

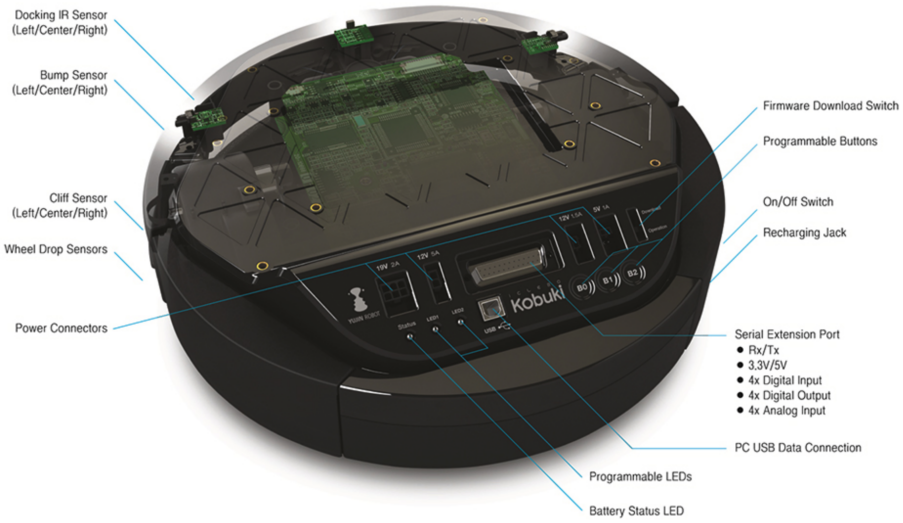

Figure 2: TurtleBot and its onboard sensors layout.[3]

Microsoft Kinect is a low cost option to equip the robot with basic RGBD imaging capabilities. Although it only provides crude depth information and is susceptible to noise in the environment, it can be a very powerful tool to perform mapping when used with appropriate algorithms.

Robot Operating System (ROS) is a commonly used robotics middleware platform. It has the functionality as an operating system, including hardware abstraction, low-level device control and message-passing between processes and package management [6].

1.4.2 Mapping Task

The competition guidelines require each team to use the Gmapping algorithm present in the ROS package to perform simultaneous localization and mapping (SLAM). GMapping is a highly efficient Rao-blackwellized particle filter to construct grid maps from laser range data and odometry [7]. Gmapping creates a occupancy map, which has black, grey and white cells. Black cell represents occupied grids, which can be the wall of the environments or the obstacles inside the environment. Grey cell represent unexplored areas, where the robot has no information about. White cell represents free area, where the robot can operate.

Although navigating to each part of the map is a big part of successding in the contest, it is not enough to obtain high quality maps. Significant drifts in odometry readings and lack of strategies to provide Gmapping with proper data to perform SLAM will result in poor map quality. Figure 4 gives a sense of the adverse effects of odometry drift if a proper SLAM algorithm (for example, the ones that support loop closure) is not used in conjunction with it.

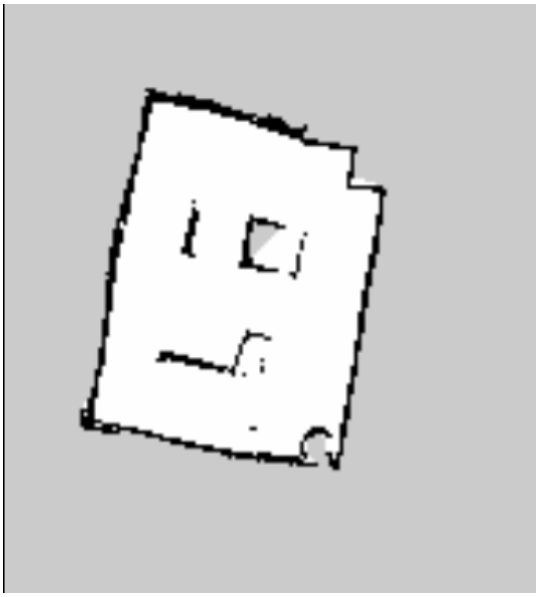

Figure 3: Example of a successful generated map. [3]

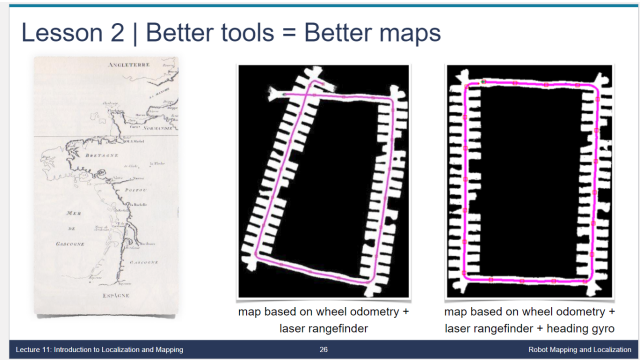

Figure 4: Wheel Odometry with laserfinder alone does not generate the best maps. [8]

2 Robot Design and Implementation

In this section we explain in detail which sensors we chose to incorporate in our design, as well as the high and low level controllers driving the behaviour of the robot.

2.1 Sensory Design

Among the many sensors present on the TurtleBot, we found that the following [1] were instrumental in designing a system that fulfilled the contest requirements. More details on the types of behaviours these sensors were used to implement can be found in Section 2.2.

2.1.1 Laser Rangefinder:

The laser rangefinder is an essential part of all of the behaviours we have implemented that involve obstacles and planning. It has sufficiently broad field of view (approximately 60 degrees) and consistently detects obstacles similar to the ones the robot will be presented on the competition day.

- **Gmapping:** Gmapping is a commonly used simultaneous localization and mapping algorithm that is part of the ROS frame work. [6]. Its implementation involves a "highly efficient Rao-Blackwellized particle filer to learn grid maps from laser range data" [9]. Therefore, laser range finder, along with wheel odometry, forms the linchpin of the mapping system.
- **Obstacle Avoidance:** We detect our robot's relative position to the obstacles it is approaching using the laser rangefinder. We first divide the range readings returned by the laser callback to three groups - "left", "center" and "right", as shown in Figure 5. If the range reading in either group falls below a threshold, we adjust the robot's orientation to avoid any collisions.
- **Obstacle Circling:** If the robot passes an obstacle on its side, this is detected by the laser rangefinder by a sudden increase in the range readings corresponding to the that side. We use this to define an "obstacle circling" behaviour, which encourages the robot to explore the interior of the map.
- **Finding empty directions:** We use laser rangefinder to move the robot towards the direction in which the range readings are largest, in order to encourage more robust exploration. The details of this is given in Section 2.2
- **Following speed constraints:** We use the range readings to detect when we get too close to an obstacle and slow down to the required speed limits.

2.1.2 Wheel Encoders/Odometry:

Signals from the wheel encoders are mostly used for mapping and low-level control.

- **Gmapping:** Encoder data is used as part of the predictive update within the Gmapping algorithm to perform SLAM.

[1] Note that since we're given a robot body already equipped with the sensors we'll need, we don't need to take size and power consumption into consideration when choosing which sensors to use.

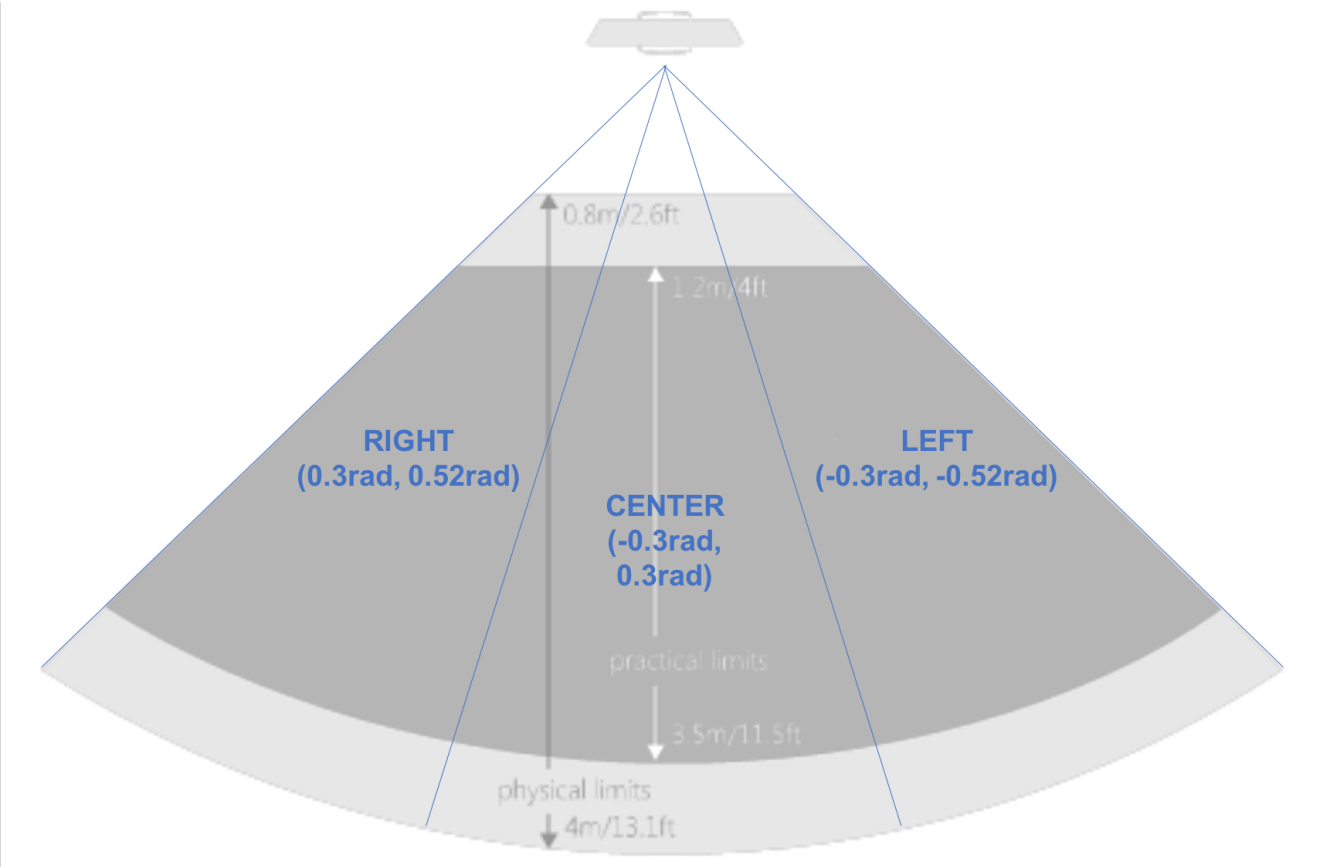

Figure 5: The range of the Kinect sensor and how we partitioned it into "left", "center" and "right". [5]

- **Heading and pose control:** We control the heading and the pose of the robot (i.e. how much we turn) using data from encoders.
- **Velocity Control:** We compute the current robot velocity using encoder data, which we use within a P controller to regulate the speed of the robot and ease speed transitions.

2.1.3 Bumper:

The bumper at the very front of the robot chasis was used for the solve purpose of backtracking from unintentional collisions with obstacles.

2.2 Controller Design

In this section, we discuss the design choices we've made regarding the high level and low level control architectures.

2.2.1 High Level Control

At an early stage in our design process, we decided to adopt a behaviour based control architecture [10], as opposed to a deliberative approach. Behaviour based robotics is based on a tight coupling between the sensory input and reflexive decision making through simple behavioural modules

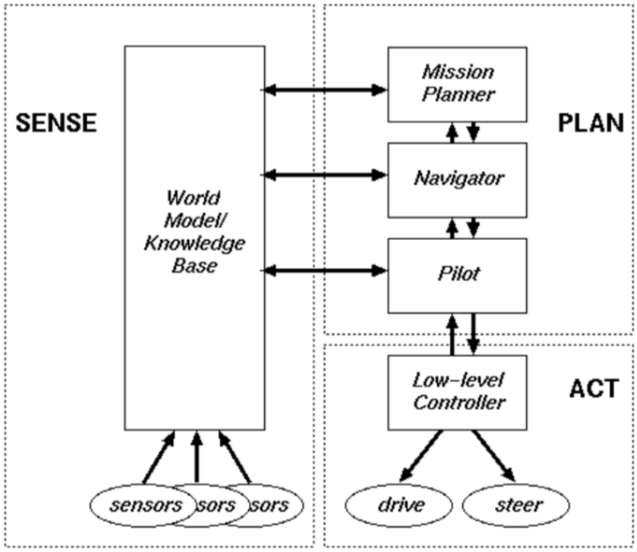

Figure 6: Deliberative architectures are hierarchical and depend on an accurate world model as well as a planning module. [3]

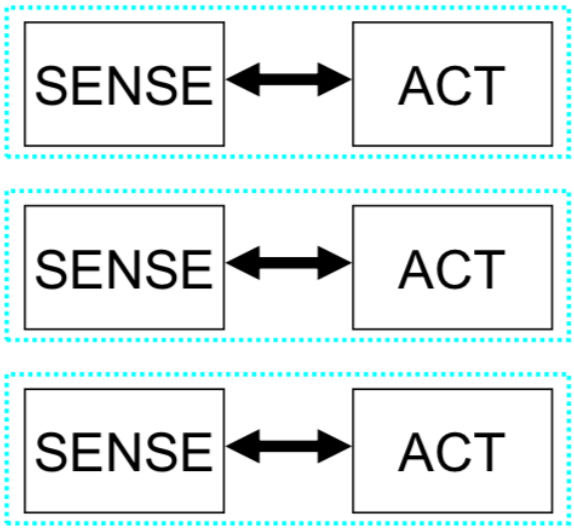

Figure 7: Reactive approaches rely on a tight coupling between sensory inputs and actions. There is no planning and holistic behaviour "emerges" from the sum of low level behaviours. [3] [10]

that are performed in parallel [3]. The deliberative approach, on the other hand, is based on a hierarchical separation of decision making into layers which do increasingly abstract reasoning as one moves from top to bottom. Figure 6 and 7 summarizes the difference between the two differing paradigms. We now list three of the reasons for why we decided on a behaviour based approach:

- **Lack of accurate world model:** In deliberative control architectures, a very accurate world model is needed in order to plan correctly. "This requires high-precision sensors and careful calibration, both of which are very difficult and expensive." [11] Navigating an unknown environment using a Kinect sensor coupled with Gmapping simply doesn't provide an accurate world model at any point during mapping. It is, on the other hand, possible to implement simple behavioural modules with local and imperfect knowledge of the world.
- **Need for robustness and generalization:** Deliberative approaches based on traditional, rigid decision making approaches are often not resilient to changes in the environment, unexpected stimulus or noise in sensor measurements. [11] Behavior based approaches don't suffer from these problems, as the aggregate behaviour of the robot (the sum of all the behavioural modules) is designed assuming imperfect knowledge of the world to begin with.
- **Need for Simplicity:** Behaviour based controllers are often simple to implement and debug. Since all the behaviours are expected to be hierarchically parallel to each other, adding a new behaviour often does not lead to catastrophic failures. This is not the case for deliberative approaches.

2.2.2 High Level Control Loop

We first summarize the holistic behaviour of the robot using pseudo-code, then describe all of the behaviours in more detail (see Algorithm 1). Please note that the pseudo-code is **not** an exact

depiction of the controller logic - we've omitted the details of behaviour coordination in favour of simplicity. Note that the more complex behaviour are only activate two minutes in the trial - this is to ensure that the boundaries of the environment are reliably navigated before the robot start exploring the inner parts of the map.

Algorithm 1: Pseudo-code for behaviour based autonomous navigation. We first pick which behaviours to choose at each spin cycle, then execute it by setting the robot's angular and translational velocity.

```
initialize_timer();
while total_time < 8 minutes do
    // Choose behaviour.
    if bumper activated then
        BACK_UP();
    else if obstacle within 0.7m away from robot then
        AVOID_OBSTACLE();
    else if full scan timer activated then
        SCAN_FULL_RANGE ;
    else if total_time > 2 minutes then
        if laser reading changed significantly on either side of robot then
            if coin toss = heads then
                CIRCLE_OBSTACLE() ;
            else
                EXPLORE_LONG_RANGE() ;
            end
        end
    end
    // Act.
    if turn then
        TURN_TO_POSE();
    else
        GO_STRAIGHT();
    end
end
```

2.2.3 List of Core Behaviours

We now elaborate on each of the behaviours mentioned in Algorithm 1.

- **GO_STRAIGHT():** This behaviour simply involves the robot to move forwards in a straight line. It is the default behaviour of the robot, unless it interacts with an obstacle.
- **BACK_UP():** If the robot hits an obstacle unintentionally, this behaviour instructs to backtrack by a fixed amount and recover from the previous position.

- **AVOID_OBSTACLE():** This behaviour gets activated if the robot gets sufficiently close to an obstacle. First, the side at which the obstacle is present is detected. Then, the robot is instructed to turn by a certain amount towards the opposite side. The amount by which the robot turns is slightly randomized in order to encourage exploration behaviour.

- **CIRCLE_OBSTACLE():** Circling around inner obstacles is a good way to cover most parts of the map. This behaviour gets activated when the robot encounters a significant increase in the range readings on either the left or right side - indicating it has just passed an inner obstacle. The robot reacts to this by turning towards that direction by a fixed amount. In order to make sure that the robot doesn't hit the obstacle it is trying to follow, a short time delay is added before the robot performs its turn. (see Figure 11)

- **EXPLORE_LONG_RANGE():** This behaviour also attempts to encourage exploratory behaviour. We considered two versions (complete and partial) of this behaviour:

  - *Complete Exploration:* In this more complex version of the behaviour, we first take a full 360°scan of the surrounding environment (Figure 12). We then filter the readings and find the predominant local maxima (Figure 13). This gives us a number of candidate directions to randomly choose from and move towards. This behaviour promises to be an efficient way to promote the exploration of all parts of the map, as long as it is initiated at the correct parts of the map during exploration. Therefore, we initiate this behaviour the same way we initiate the CIRCLE_OBSTACLE() behaviour: immediately after moving past obstacles.
  - *Partial Exploration:* This behaviour is a simpler version of the above. Rather than taking a full scan and identifying local maxima, we just process the laser range readings already in front of us, and move towards the global minima. This behaviour is also initiated immediately after an obstacle has been passed.

  Due to the need to tune the filtering process and limited testing time, we decided to incorporate only the partial exploration behaviour. We list the complete exploration behaviour as one of the promising directions to pursue to improve the mapping quality.

2.2.4 Coordinating Behaviours in Parallel

We've implemented a simple, priority based coordination mechanism to resolve potential conflicts between behaviours. Only behaviours with higher priority can interrupt those with lower priority when triggered. The priority ordering (from high to low) of our behaviours are as follows: BACK_UP, AVOID_OBSTACLE, CIRCLE_OBSTACLE, EXPLORE_LONG_RANGE, GO_STRAIGHT.

2.3 Low Level Control

The TurtleBot is already equipped with low level controllers which turn our translational and angular velocity commands into motor signals. Therefore, the challenge on our part was to send smooth and realizable reference signals to TurtleBot's controllers so that the motion of the robot was smooth. To this end, we wrapped all of the reference signals with a "virtual P controller", which smoothens out abrupt changes in signal.

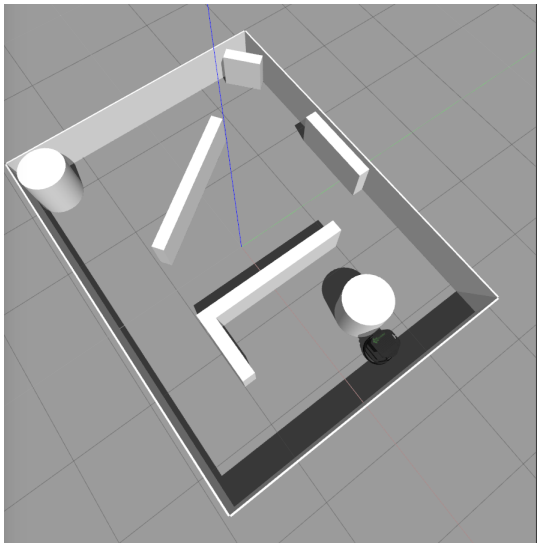

Figure 8: Detected end of obstacle.

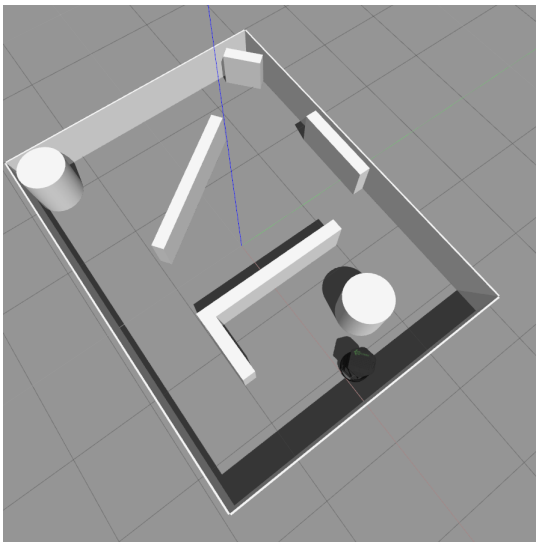

Figure 9: Moved right past obstacle.

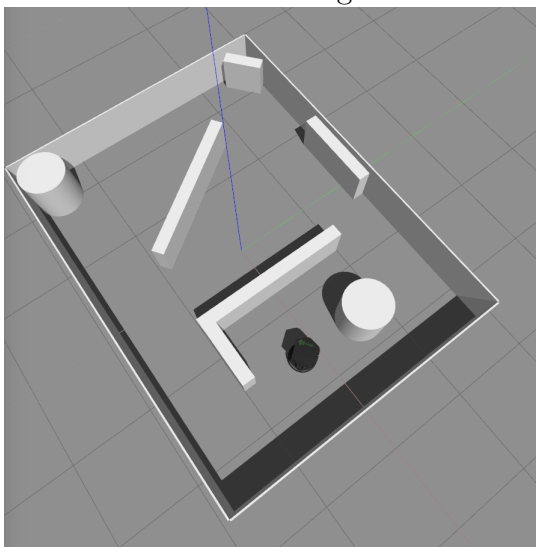

Figure 10: Turned towards obstacle.

Figure 11: **Simulation of the obstacle circling behaviour** The obstacle avoidance algorithm alone does a great job at circling besides the outer walls of the environment, but has difficulty navigating inside. The obstacle circling behaviour ensures that the interior of the map can be explored by encouraging the robot to move around the inner obstacles.

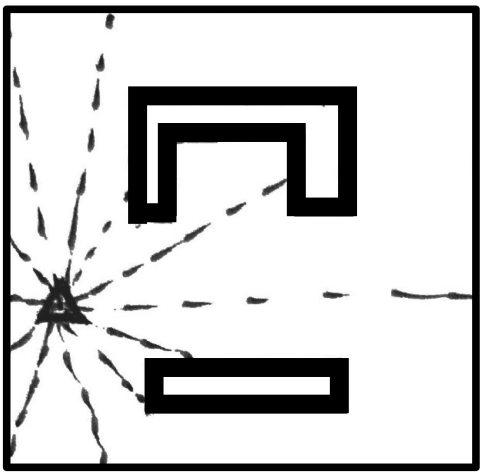

Figure 12: **Step 1 of Complete exploration:** Get a full scan of the surrounding and get range measurements .

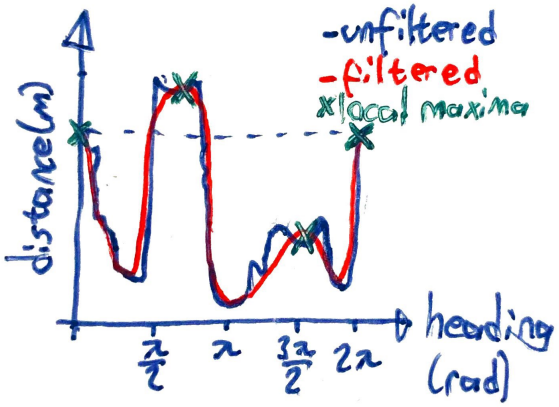

Figure 13: **Steps 2 and 3 of Complete Exploration:** Obtain a graph of all the range measurements, filter out the noise and detect the local maxima.

3 Strategy

In this section, we cover our design process to outline the reasoning behind the engineering decisions we've made along the way.

3.1 Design Process

The behaviour based high level control architecture comes with a big advantage: it makes it possible to implement and test each behaviour independent from each other in a modular fashion. This was the guiding principle in our design process. We first started with building the simplest system that did a satisfactory job in building a map, then gradually improved the performance with more complex behaviours.

3.1.1 Step 1: Minimum Viable Design

Before attempting more complicated solutions, we first implemented two simple behaviours: **go forward** and **turn away from an obstacle when faced with one**. After minimal tuning, these behaviours were sufficient to reliably navigate around the outer walls of the contest environment.

3.1.2 Step 2: Adding Increasingly Complex Behaviours:

To equip the minimum viable design with the ability to explore the center of the contest environment, we implemented two additional behaviours: textbfobstacle circling and **long range exploration**. These behaviours were sufficient to encourage the robot to explore different parts of the map in a stochastic fashion.

3.1.3 Step 3: Tuning:

In order to improve the mapping quality and the efficiency exploration, we fine-tuned our design with the following additions:

- **Scanning surroundings** Every once in a while, we instructed the robot to stop and scan its surrounding 360°environment. This provides the Gmapping algorithm with the opportunity to refine its belief on where the nearby obstacles are.
- **Velocity control:** We implemented a simple P controller to smooth out the abrupt motion of the robot when it stopped and started moving to prevent encoder slipping.
- **Stopping briefly after moving:** We've noticed that the robot had a bias to move in arcs towards the direction it has recently completed turning, even though it was instructed to go straight. To eliminate the adverse effect of this on the mapping quality, we added a short stopping period right after completing turns, which significantly mitigated the problem.

3.2 Picking Combinations of Behaviours

Once we had all the behaviours implemented, we ran several simulations to observe the relative effect of each behaviour on the system. Our experiments showed that the EXPLORE_LONG_RANGE behaviour itself was sufficient to promote robust exploration, while having the least number of potential failure cases. Therefore, we favoured the simplest and most robust solution and decided to

remove the obstacle following behaviour from our algorithm for the first take in the contest, and save the hybrid behaviour for the second take, if there is an unexpected failure case.

3.3 From Course Grain to Fine Grain

We've noticed that the minimum viable design described above did a very good job reliably navigating the outer boundaries of the map without any more complicated behaviour. In order to exploit this, we set our algorithm to activate more complex behaviour only after 2 minutes have passed. This way, we guarantee that we obtain a satisfactory map of the environment before potential failure cases inside the environment.

4 Future Recommendations

There are a variety of ways to improve the current system to achieve better map qualities in shorter time.

4.1 Tuning Implementation Parameters

In a behaviour based approach where decisions about current actions are made stochastically, it is essential to tune the design parameters. The very first step to improve our existing design would be to spend more time on the testing track and test a variety of different design parameters.

4.2 Using Complete Long Range Exploration Behaviour

As discussed in Section 2.2.3, we haven't incorporated the complete long range exploration behaviour in our system due to limited testing time. We regarding this as a promising future direction to take in order to guarantee that our system will explore all the parts of the environment.

4.3 Balancing Behaviour and Deliberation

As we've discussed before, a completely behaviour based control architecture is ideal to achieve robustness against unexpected inputs and limited sensor quality/mapping algorithms. However, the map used in the competition is small and static enough to also justify a more deliberative approach [2]. A high level decision making module, which has access to the map constructed so far, can guide the robot to the unexplored parts of the map using efficient search algorithms such as A* or maximum information gain [12], while successfully avoiding all the obstacles in the environment. What makes this approach particularly difficult is that evidence grids outputted by gmapping change quite significantly over time and don't incorporate the uncertainty associated with the occupancy prediction for each grid. This makes deliberative approaches which output way-points for a lower level controller to follow quite unreliable and difficult to implement.

A more sophisticated decision making algorithm will guide the higher level planning of the robot through the use of the map that has been constructed so far.

[2] Given one is willing to spend enough time to make sure the deliberative approach is robust enough

5 Contributions of Team Members

Each team member played an integral part throughout the entire design process and have meaningfully contributed to each task of the competition. Overall, we conclude that each team member have contributed equally to the final design. Solely to comply with the report guidelines, we now outline some of the specific contributions of the team members.

The majority of the C++ implementation were carried out by Xuchan Bao and Jingxing Qian. Zihan Wang, Zheng Yao and Cem Anil have made significant contributions to designing the high level behavioural control architecture. Cem Anil undertook the majority of the writing of this report, with significant inputs from Zheng Yao and Zihan Wang. Jingxing Qian's and Xuchan Bao's common sense and extensive practical knowledge in robotics was essential in making quick progress without getting stuck with dead ends and picking out good ideas from impractical ones.

Some of the work we did also didn't make its way to the final design. Xuchan Bao made significant progress in implementing a deliberative, map based decision making algorithm and had to abandoned that path due to limited test time. For similar reasons, the complete long range exploration behaviour Zheng Yao and Cem Anil were promoting wasn't integrated in the final design.

References

[1] Robin R Murphy, Satoshi Tadokoro, Daniele Nardi, et al. "Search and rescue robotics". In: *Springer handbook of robotics*. Springer, 2008, pp. 1151–1173.

[2] Bryan Chmura, Robert N McKee, Victor Younger, et al. *Robotic vacuum with removable portable vacuum and semi-automated environment mapping*. US Patent 7,113,847. 2006.

[3] Silas Alves Godie Nejat. *Mechatronics Systems: Design and Integration*. Jan. 2019.

[4] Willow Garage. "Turtlebot". In: *Website: http://turtlebot. com/last visited* (2011), pp. 11–25.

[5] Zhengyou Zhang. "Microsoft kinect sensor and its effect". In: *IEEE multimedia* 19.2 (2012), pp. 4–10.

[6] Morgan Quigley, Ken Conley, Brian Gerkey, et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[7] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. "Improved techniques for grid mapping with rao-blackwellized particle filters". In: *IEEE transactions on Robotics* 23.1 (2007), pp. 34–46.

[8] Steven Waslander. *Mobile Robotics*. Jan. 2019.

[9] Hesham Ibrahim Mohamed Ahmed Omara and Khairul Salleh Mohamed Sahari. "Indoor mapping using kinect and ROS". In: *Agents, Multi-Agent Systems and Robotics (ISAMSR), 2015 International Symposium on*. IEEE. 2015, pp. 110–116.

[10] Rodney A Brooks. "Intelligence without representation". In: *Artificial intelligence* 47.1-3 (1991), pp. 139–159.

[11] MARC CARRERAS I PÉREZ and JB Grabulosa. "An overview of Behavioural-based Robotics with simulated implementations on an Underwater Vehicle". In: *Institut d'Informàtica i Aplicacions Universitat de Girona, Girona, PhD Thesis* (2000).

[12] Cyrill Stachniss and Wolfram Burgard. "Exploring unknown environments with mobile robots using coverage maps". In: *IJCAI*. 2003, pp. 1127–1134.

Appendix A Full C++ ROS code

We include the full C++ ROS code in the following listing. This is the content of 'contest1.cpp', the only file that contains our source code.

```
#include <ros/console.h>
#include "ros/ros.h"
#include <tf/tf.h>
#include <geometry_msgs/Pose.h>
#include <geometry_msgs/Twist.h>
#include <nav_msgs/Odometry.h>
#include <kobuki_msgs/BumperEvent.h>
#include <sensor_msgs/LaserScan.h>
#include <eStop.h>

#include <stdio.h>
#include <iostream>
#include <cmath>
#include <chrono>
#include <random>

using namespace std;


class Timer {
public:
    // Default constructor
    Timer() {}

    //Start the timer
    void begin() {
        _begin = std::chrono::steady_clock::now();
    }


    //Stop the timer
    void stop() {
        _end = std::chrono::steady_clock::now();
    }


    //Get the amount of time passed from start to stop in
       milliseconds
    double getDuration() {
        auto span = std::chrono::duration_cast<std::chrono::
           nanoseconds>(_end - _begin);
        return span.count() / 1000000.0;
    }
```

```
private:
    std::chrono::steady_clock::time_point _begin;
    std::chrono::steady_clock::time_point _end;
};


class Pose2 {
public:
    // Constructors
    Pose2() : _x(0), _y(0), _yaw(0), _velocity(0) {}

    Pose2(double x, double y, double yaw) : _x(x), _y(y), _yaw(
       wrapAngle(yaw)) {}

    Pose2(geometry_msgs::Pose p) : Pose2(p.position.x, p.position.y
       , tf::getYaw(p.orientation)) {}

    // Operators
    Pose2 operator+(const Pose2 &innovation) {
        double result_x = innovation.x() * cos(yaw()) - innovation.
           y() * sin(yaw()) + x();
        double result_y = innovation.x() * sin(yaw()) + innovation.
           y() * cos(yaw()) + y();
        double result_yaw = innovation.yaw() + yaw();

        return Pose2(result_x, result_y, result_yaw);
    }

    Pose2 operator-(const Pose2 &innovation) {
        double theta = innovation.yaw();
        double diff_x = x() - innovation.x();
        double diff_y = y() - innovation.y();

        double result_x = diff_x * cos(theta) + diff_y * sin(theta)
           ;
        double result_y = -diff_x * sin(theta) + diff_y * cos(theta
           );
        double result_yaw = yaw() - theta;

        return Pose2(result_x, result_y, result_yaw);
    }

    bool operator==(const Pose2 other) {
        return fabs(x() - other.x()) < 1e-10 && fabs(y() - other.y
           ()) < 1e-10 && fabs(yaw() - other.yaw()) < 1e-10;
```

```
    }

    // Print function
    void print(std::string str = "") {
        std::cout << str << "␣x:␣" << x() << "␣␣␣␣" << "y:␣" << y()
            << "␣␣␣␣" << "yaw:␣" << yaw() << std::endl;
    }

    void setYaw(double yaw) {
        _yaw = wrapAngle(yaw);
    }

    // Get funtions
    const double &x() const {
        return _x;
    }

    const double &y() const {
        return _y;
    }

    const double &yaw() const {
        return _yaw;
    }

    double norm() const {
        return sqrt(x() * x() + y() * y());
    }

    geometry_msgs::Pose toMsg() const {
        geometry_msgs::Pose p;
        p.position.x = x();
        p.position.y = y();
        p.position.z = 0;
        p.orientation = tf::createQuaternionMsgFromYaw(yaw());
        return p;
    }

private:
    double wrapAngle(double angle) {
        if (angle <= (double) -M_PI) return wrapAngle(angle + 2.0 *
            (double) M_PI);
        if (angle > (double) M_PI) return wrapAngle(angle - 2.0 * (
           double) M_PI);
        return angle;
    }
```

```
    double _x;
    double _y;
    double _yaw;

    double _velocity;
};

double min(const std::vector<float> &vec, size_t start, size_t end)
    {
    double curr_min = std::numeric_limits<float>::max();
    for (size_t i(start); i < end; i++) {
        if (!isnan(vec[i]) && vec[i] < curr_min) {
            curr_min = vec[i];
        }
    }
    return curr_min;
}

double max(const std::vector<float> &vec, size_t start, size_t end)
    {
    double curr_max = -std::numeric_limits<float>::max();
    for (size_t i(start); i < end; i++) {
        if (!isnan(vec[i]) && vec[i] > curr_max) {
            curr_max = vec[i];
        }
    }
    return curr_max;
}

size_t max_idx(const std::vector<float> &vec, size_t start, size_t
   end) {
    float curr_max = 0;
    size_t max_id = start;
    for (size_t i(start); i < end; i++) {
        if (!isnan(vec[i]) && vec[i] > curr_max) {
            curr_max = vec[i];
            max_id = i;

        }
    }
    return max_id;
}

// Global variables, used to record the state of the system.
int g_dir = 1;
double g_back_up_duration = 1.0;
double g_progress = 0.0;
```

```
bool g_do_turn = false;
bool g_got_odom = false;
bool g_back_up = false;
Pose2 g_current_pose;
Pose2 g_desire_pose;
std::vector<int> g_bumper_states = {0, 0, 0, 0, 0};
Timer g_timer;

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<> dis(0.1, 0.6);

double g_min_reading = 100;

double prev_left_reading = 100;
double prev_right_reading = 100;

double threshold_side_obstacle = 0.7;

int side_obstacle_turning_delay = 33;
int current_delay_cycles = 0;
bool delay_turn = false;
bool g_do_turn_side_obstacle = false;
Timer g_timer_side_obstacle;
double interval_until_next_side_obs_turn = 5.0;
Timer g_timer_after_turn_before_straight;
double interval_after_turn_before_straight = 1.0;

Timer g_timer_turn_around;
double interval_turn_around = 90.0;
bool g_turning_around = false;
bool g_turned_half_way = false;

double g_velocity = 0;
double kp_vel = 0.4;

float deviate_to_max_range(const std::vector<float> &ranges, float
   max_angle, float min_angle, float dangle) {
    size_t max_id = max_idx(ranges, 0, ranges.size());
    std::cout << max_id << std::endl;
    float angle_of_max_range = min_angle + dangle * max_id;
    return angle_of_max_range - (max_angle + min_angle) / 2;
}


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

```
    g_bumper_states[msg->bumper] = msg->state;
}


void set_turn_angle(int turn_dir, double turn_dyaw) {
    g_do_turn = true;
    g_desire_pose = g_current_pose + Pose2(0, 0, turn_dir *
       turn_dyaw);
}


void laserCallback(const sensor_msgs::LaserScan::ConstPtr &msg) {
    if (g_turning_around) {
        std::cout << "Turning around, do nothing in LaserCallback"
           << std::endl;
        return;
    }

    double max_angle = msg->angle_max;
    double min_angle = msg->angle_min;
    double dangle = msg->angle_increment;
    auto ranges = msg->ranges;
    size_t n = ranges.size();
    double check_max = 0.3;
    size_t start_idx = floor((max_angle - check_max) / dangle);
    size_t mid_idx = floor(n / 2);

    double threshold = 0.7;

    double center_min = min(ranges, start_idx, n - start_idx);
    double right_min = min(ranges, 0, start_idx);
    double left_min = min(ranges, n - start_idx, n - 1);

    g_min_reading = min(ranges, 0, n - 1);

    if (center_min < threshold) {
        if (!g_do_turn) g_dir = left_min > right_min ? 1 : -1;
        double dyaw = 0.3;
        if (g_progress > 0.4) {
            dyaw = double(dis(gen));
        }
        set_turn_angle(g_dir, dyaw);
        delay_turn = false;
        std::cout << "Reset obstacle following" << std::endl;
    } else {
        if (g_progress > 0.25) { // only try following obstacle
           beyond certain progress value
```

```
            if (delay_turn) {
                if (current_delay_cycles <
                   side_obstacle_turning_delay) {
                    std::cout << "Delaying␣turning␣" << g_dir << "
                       ...␣current␣cycle␣=␣" <<
                       current_delay_cycles
                              << std::endl;
                    current_delay_cycles++;
                } else if (!g_do_turn_side_obstacle) {
                    g_timer_side_obstacle.stop();
                    std::cout << "Time␣since␣last␣obs␣turn␣=␣" <<
                       g_timer_side_obstacle.getDuration() / 1000.0
                              << std::endl;
                    if (g_timer_side_obstacle.getDuration() /
                       1000.0 > interval_until_next_side_obs_turn)
                       {
                        current_delay_cycles = 0;
                        std::cout << ">>>␣Moving␣past␣obstacle..."
                           << std::endl;
                        // Two options for turning. Default to max
                           range.
                        // Change "true" to "false" for obstacle
                           following.
                        if (true) { // Turn towards the direction
                           with largest distance (max range)
                            double dyaw = deviate_to_max_range(
                               ranges, max_angle, min_angle, dangle
                               );
                            set_turn_angle(1, dyaw);
                        } else { // Turn a fixed angle to follow
                           the obstacle
                            std::cout << "\n>>>␣Moving␣past␣
                               obstacle...\n\n";
                            double dyaw = 0.9; // ~45 degrees
                            set_turn_angle(g_dir, dyaw);
                        }
                        g_do_turn_side_obstacle = true;
                    }
                }
            } else { // Detecting reading changing beyond threshold
                => start countdown for turning
                if (left_min - prev_left_reading >
                   threshold_side_obstacle) {
                    std::cout << ">>>␣Hallucinating␣turning␣LEFT...
                       " << std::endl;
                    g_dir = 1;
                    delay_turn = true;
```

```
                    current_delay_cycles = 0;

                } else if (right_min - prev_right_reading >
                   threshold_side_obstacle) {
                    std::cout << ">>> Hallucinating turning RIGHT
                       ..." << std::endl;
                    g_dir = -1;
                    delay_turn = true;
                    current_delay_cycles = 0;
                }
            }

        }
        prev_left_reading = left_min;
        prev_right_reading = right_min;
    }
}

void odomCallback(const nav_msgs::Odometry::ConstPtr &msg) {
    g_got_odom = true;
    g_current_pose = Pose2(msg->pose.pose);
    g_velocity = msg->twist.twist.linear.x;
}

int sum(const std::vector<int> &vec) {
    int s = 0;
    for (const auto &e : vec) {
        s += e;
    }
    return s;
}

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

    ros::init(argc, argv, "image_listener");
    ros::NodeHandle nh;
    teleController eStop;

    ros::Subscriber bumper_sub = nh.subscribe("mobile_base/events/
       bumper", 10, &bumperCallback);
    ros::Subscriber laser_sub = nh.subscribe("scan", 1, &
       laserCallback);
    ros::Subscriber odom_sub = nh.subscribe("odom", 1, &
       odomCallback);

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

```
    double angular = 0.0;
    double linear = 0.0;
    geometry_msgs::Twist vel;

    ros::Rate rate(50);

    Timer back_up_start;
    g_timer.begin();
    g_timer_side_obstacle.begin();
    g_timer_turn_around.begin();

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

    while (ros::ok() && seconds_elapsed <= 480) {
        g_timer.stop();
        double elapse = g_timer.getDuration() / 1000.0;
        g_progress = elapse / double(480.0);
//        std::cout << "Progress: " << g_progress * 100.0 << "%" <<
    std::endl;

        ros::spinOnce();
        //.....**E-STOP DO NOT TOUCH**.......
        eStop.block();
        //....................................

        if (!g_got_odom) continue;

        if (sum(g_bumper_states) > 0) {
            g_back_up = true;
            delay_turn = false;
            back_up_start.begin();
        }

        back_up_start.stop();
        if (g_back_up && (back_up_start.getDuration() / 1000.0 <
           1.5)) {
            linear = -0.2;
            angular = 0;
        } else if (g_back_up && (back_up_start.getDuration() /
           1000.0 < 1.0)) {
            g_back_up = false;
        } else {
            if (g_do_turn) {
```

```
            double delta_yaw = (g_desire_pose - g_current_pose)
               .yaw();
            angular = delta_yaw / fabs(delta_yaw) * 0.3;
            linear = 0.0;
            if (fabs(delta_yaw) < 0.02) {
                g_do_turn = false;
                if (g_do_turn_side_obstacle) {
                    g_timer_side_obstacle.stop();
                    g_timer_side_obstacle.begin();
                    std::cout << "Finished side obs turning" <<
                        std::endl;
                    g_do_turn_side_obstacle = false;
                }
            }
            g_timer_after_turn_before_straight.stop();
            g_timer_after_turn_before_straight.begin();
        } else {
            g_timer_after_turn_before_straight.stop();
            // Stop briefly just after turning, to avoid
               traveling in arcs
            if (g_timer_after_turn_before_straight.getDuration
               () / 1000.0 >
               interval_after_turn_before_straight) {

                // Check if we should turn around 360 degrees
                   now
                g_timer_turn_around.stop();
                if (!g_turning_around && g_timer_turn_around.
                   getDuration() / 1000.0 >
                   interval_turn_around) {
                    // Start turning around 360 degrees
                    std::cout << std::endl
                              << "==================\nStart
                                 turning around 360 degrees\n
                                 ====================="
                              << std::endl;
                    g_turning_around = true;
                    set_turn_angle(1, 3.1);
                    g_turned_half_way = false;
                } else if (g_turning_around) { // After
                   finishing turning around 360 degrees
                    if (!g_turned_half_way) { // just finished
                       turning half way
                        set_turn_angle(1, 3.1);
                        g_turned_half_way = true;
                    } else {
                        g_timer_turn_around.stop();
```

```
                            g_timer_turn_around.begin();
                            g_turning_around = false;
                        }

                    }
                    linear = g_min_reading > 0.6 ? 0.18 : 0.1;
                    angular = 0.0;
                } else {
                    linear = 0.0;
                    angular = 0.0;
                }
            }
        }

        //
        vel.angular.z = angular;

        vel.linear.x = g_velocity + kp_vel * (linear - g_velocity);

        vel_pub.publish(vel);

        rate.sleep();

        seconds_elapsed = std::chrono::duration_cast<std::chrono::
           seconds>(
                std::chrono::system_clock::now() - start).count();
    }

    return 0;
}
```

Listing 1: Listing of the full ROS C++ code.