Turtlebot Maze Navigation

Olivia Adams, Swetha Pallerla, Kevin Yen, Kush Patel

University of Maryland

ENAE450

I. Introduction

The goal was to implement an algorithm in ROS2 to autonomously navigate a Turtlebot3 Waffle Pi robot out of a maze (Figure 1) using lidar data to avoid obstacles. There were two portions to this: a real-world run ("hardware"), and a simulation in Gazebo ("simulation").

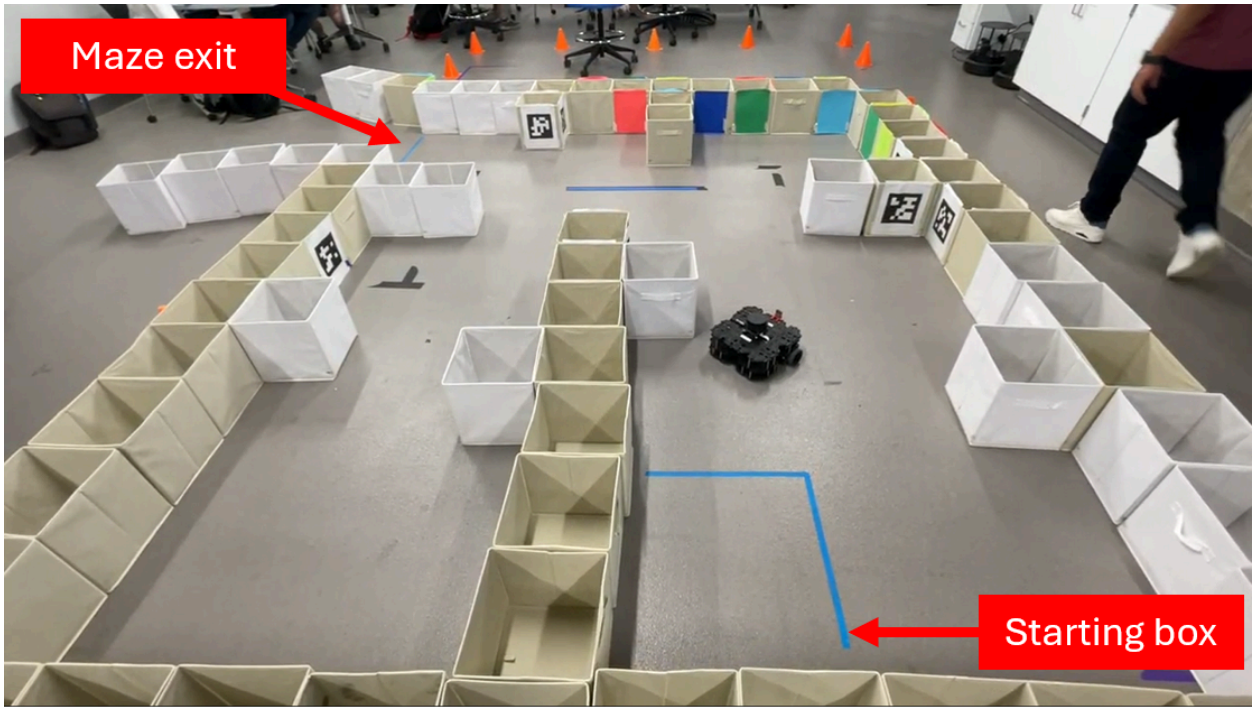

Figure 1 (above): The maze to be navigated. Each cube is approximately 1ft x 1ft x 1ft. The blue tape indicates the starting box. The Turtlebot is always placed in this box before beginning an attempt to escape the maze. The exit of the maze is indicated in the top left corner with a clear bound marked by blue tape for timing purposes.

For the hardware section, we explored the relevant topics and the message types that were published to those topics (Table 1). We created a package called maze_hardware with a script called maze_nav.py. In this script, a node called Move was created. This node had a subscriber subscribed to topic scan with LaserScan messages, and a publisher publishing to topic cmd_vel with Twist messages. Each time a LaserScan message was received to the subscriber, a callback would invoke a navigation routine. The navigation routine took the LaserScan message as an input, and based on where around the robot had the farthest distance, the routine used the publisher to publish a Twist message to topic cmd_vel to set the linear and angular velocities of the Turtlebot to navigate. To connect to the TurtleBot, we first connected our virtual machine (VM) to the same network as the TurtleBot, then accessed it remotely via SSH.

Message Type	Topic	Message Definition
Twist	cmd_vel	Vector3 linear Vector3 angular
LaserScan	scan	std_msgs/msg/Header header float angle_min float angle_max float angle_increment float time_increment float scan_time float range_min float range_max float[] ranges float[] intensities

Table 1 (above): Message types, the topic they are published to, and the definition of the message.

For the simulation section, a maze_simulation package was created. Within the package, the maze model and config files were imported into separate folders within the models folder. Then, with the empty TurtleBot3 world file obtained from the example TurtleBot3 packages as a base, the mazes were added to the world file so that they would be imported as a part of the world along with the TurtleBot when launched. Next, the navigation script from the hardware section was added into the package. Finally, a launch file was created to launch the world file and the node file, allowing the entire simulation to be run with one terminal command. With testing, some changes were made to the navigation algorithm to better suit it for the maze simulation.

II. Hardware

II.A. Methods

The maze navigation algorithm we decided to use is called “zone nav”. The general idea was to collect lidar scans from the front of the robot and group them into three “buckets” : left, center, and right. Figure 2, down below, shows the span of each of these buckets. We found the average distance between the robot and any obstacles for each of the bucket regions. Then we follow a simple decision structure. If the average distance in the center region was shorter than our threshold, then we stop the robot since there is an obstruction right in front of us. Then we would compare our left and right values and turn towards the side with more space. Otherwise when the front was clear, we would compare all three zones and move forward while veering towards the direction with most space. This function was repeatedly called in scan_callback()

which received the LaserScan message, so our navigation was dynamically adjusting to new lidar data.

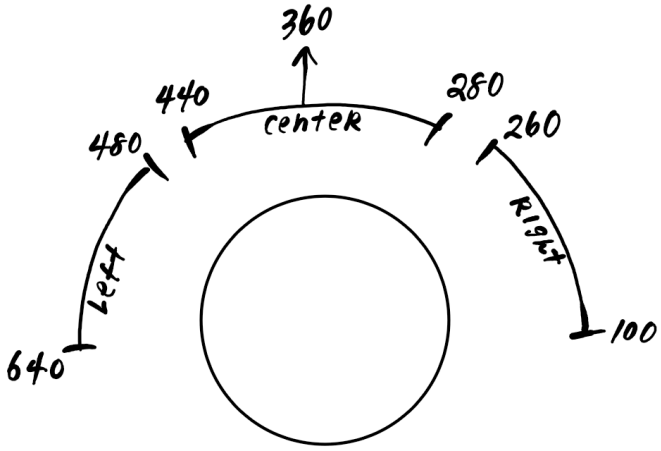

Figure 2 (above): Indexes describing the bounds of the left, center, and right buckets.

In order to ensure that our lidar indexing was accurate, we created a function called get_min_distance_in_range(). We would send the entire range of indexes to the function and place a pencil at the front of the robot, then we would look at the output on the terminal to see which index had the smallest distance value indicating the front. Later on, we used the same function to determine how wide we wanted our field of view for the navigation to be. We wanted it to be wide enough that the turtlebot would be able to see its peripherals but also not too wide that it would be distracted by obstacles that were out of our path. After some testing with get_min_distance_in_range() and zone_nav(), we decided on a 80 degrees of visibility for each direction.

II.B. Results

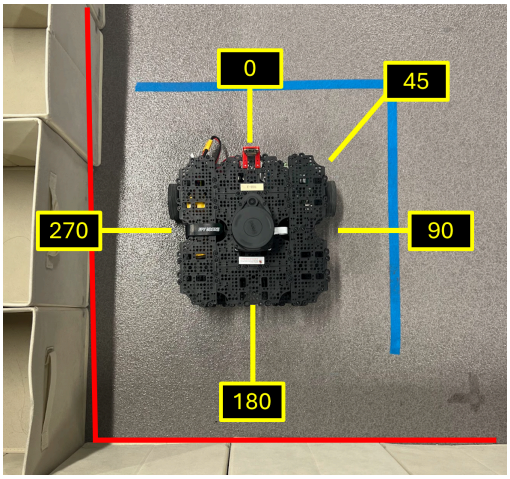

Figure 3 (above): Starting angle.

Trial #	Starting angle [deg]	Linear velocity [m/s]	Angular velocity [rad/s]	Runtime [s]	Collisions w/ wall
1	0	0.1	0.3	54.78	0
2	45	0.1	0.3	53.13	0
3	90	0.1	0.3	55.7	0
4	45	0.26	1.0	21.73	0
5	0	0.26	1.0	21.0	0
6	90	0.26	1.0	35.0	No collisions, but odd behavior
7	0	0.26	1.0	30.0	0

Table 4 (above): The results of 7 trials using the zone_nav routine. Trials 1-3 were run at low linear and angular velocity settings. Trials 4-6 were run at the maximum linear velocity and a higher angular velocity.

Trial #	Video link
1	https://drive.google.com/file/d/1VS2o7CvZ6le_t6O1XzCfHZbfBn7PlDx2/view?usp=sharing
2	https://drive.google.com/file/d/1ItgLEVnLrlIifTR68P4LHKQJjsAiz2aq/view?usp=sharing
3	https://drive.google.com/file/d/1Yb0Pa7JOjwTd3mKZBRRQP2rs1EAFkJd7/view?usp=drive_link
4	https://drive.google.com/file/d/1z4nYNis5QqSrOh02eD2W-g8RA96CDi4g/view?usp=drive_link
5	https://drive.google.com/file/d/1rWSndhr2vwRIzDUksr48beBzTWyy_PEq/view?usp=drive_link
6	https://drive.google.com/file/d/1fshrWuvUE83TyPpqkPdDPy2YRv5y_VED/view?usp=drive_link
7 (new maze)	https://drive.google.com/file/d/1h_i7OgbnkHf3F83CUsey0Qw_pvDVgc9T/view?usp=sharing

Table 5 (above): Links to videos of each trial.

Trials 1-3 all performed as expected with the robot navigating out of the maze with no collisions.

Trials 4-5 were very successful and produced times that were competitive with the best trials in the class at the time.

Trial 6 with a 90 degree starting angle began with an unexpected right turn towards the close right wall with a nonzero linear velocity lower than 0.26 m/s by observation. It is predicted that this was due to the navigation routine switching very quickly between the navigation settings for when it sees a front collision risk (0 m/s linear, 1 rad/s angular) and for when it does not see a front collision risk (0.26 m/s linear, 0 rad/s angular), producing a lower average speed. The cause of this switching behavior is unknown, as Trial 3 had the same starting angle and did not replicate this issue. Additionally, the left side of the robot clearly had more open space than the right side of the robot, and the reason for the robot heading closer to the right wall is unknown.

Trial 7 was run using the same velocity settings in a different maze. The robot did not perform ideally in Trial 7: it took turns that it should not have taken. This may be influenced by the lack of fidelity due to the low number of zones. Using averages of only 3 zones can result in the actual farthest location around the robot to be ignored. For example, if the left zone has the a very far open space occupying a small angular region but otherwise has a very close wall that brings down the average, and the right zone has a somewhat far wall occupying the entire angular space resulting in a medium average, the robot will turn towards the right zone because the average is higher, instead of turning towards the zone with the very far open space. This problem would be solved with increasing the number of buckets so that far distances in small angular regions will hold more weight instead of getting “drowned out” by many closer values bringing down the average. Despite this, the robot did navigate out of the maze, showing that the algorithm is robust enough to accommodate other maze configurations.

Overall, the navigation routine was robust enough to accommodate 3 different starting angles and two maze configurations with 0 wall collisions with at least 2 highly competitive finish times.

III. Simulation

III.A. Methods

For the simulation, we started with the base navigation algorithm utilized for the hardware section. Due to the nature of the Gazebo maze in comparison to the real world maze, some changes were made. One change was the removal of code that would drive the robot in an arc to the left or right if they were open. This change was necessary because the simulation maze walls are significantly farther apart than in the real world maze walls, so the robot would often drive in a circle in a single hallway. With this change, the robot will now drive in a straight line

until it detects an obstacle ahead. Then, based on the distances to the left or right, it will turn away from the obstacle. We ran into some issues where the robot would get caught in a loop facing the thin edge of a wall, where it would turn a little to the right, try to move forward, then stop and turn a little to the left, and continue the cycle forever. To fix this, we split the center bucket into a center left and a center right bucket, and checked the average distance of both buckets. If the average distance was greater than the minimum obstacle distance, we would turn in that direction. Another change was the adjustment of the LIDAR angles used for the buckets. In the simulation, the 0 angle is the front, and the LIDAR only takes samples in 1 degree increments, as opposed to the 0.5 degree increments used by the real world robot. Finally, code was added to check if the front, left, and right sides of the robot were clear, to determine when the robot had exited the maze. Once this condition was true, the robot would stop, then turn at 1 rad/s for 3.14 seconds to turn 180 degrees, then stop again and ignore all movement commands.

III.B. Results

Our simulation time for maze 2 was 1:44. The video of this run can be found at https://drive.google.com/file/d/1nAQ01QgfkOrwUktHPSC93DhT0uBYtJG3/view?usp=drive_link. The robot performed as expected, driving in straight lines until it met a wall, then turning in the direction of the greater open space.

IV. Retrospect

IV.A. Hardware

During the software development phase, we explored a few different maze algorithms – all with different levels of success.

1. The first algorithm was a simple wall detection program. We would set the robot on a straight path. When an obstacle was detected within our threshold level, we would scan our left and right sides and turn 90 degrees towards the side with more space. While this algorithm got us through a simple L shaped maze, we ran into a few limitations. The robots' turns were never exactly 90 degrees, always a little more or less, this set our path on a skew and over long distances the turtlebot would veer into walls. This issue was further exacerbated by the walls of the maze not being completely straight. If we were to continue with this algorithm, we would expand the logic into a more sophisticated wall following design with more edge case and implement a PD control algorithm that would help turtlebot maintain a steady distance from the wall. We decided that the time it would

take to implement and debug wall following and edge cases was too long to make it in time for the competition.

2. The second algorithm we tried was called “vector nav”. I found this on a robotics forum from a man who was trying to navigate through an organic path with many irregular curves. The gist of this algorithm was to find the direction with the most space and drive towards it. To do this, we divide the front of the robot into three partitions (left, center, right) and find the average lidar distance scan in each. Then we would convert the distances into vectors that represented the average distance to an obstacle on each of the sides. After adding the vectors together, the turtlebot would have a direction to drive towards. Throughout the run, this algorithm was constantly called so that our direction vector could be updated. While the idea was simple, we had many difficulties implementing it. The result of our math produced a vector (magnitude and angle) but our drive data type (twist) takes a linear and angular velocity. Ideally the function is called every time the lidar is scanned in scan_callback, but because we require an angular velocity we had to wait for the robot to fully turn before calling the function again. I tried to call vector nav in a separate function that ran every second, but I found trying to copy the lidar msg outside of the scan_callback to be difficult to do and caused division by zero errors. This resulted in a weird timing and latency problem that ultimately resulted in the turtlebot exhibiting unpredictable behavior

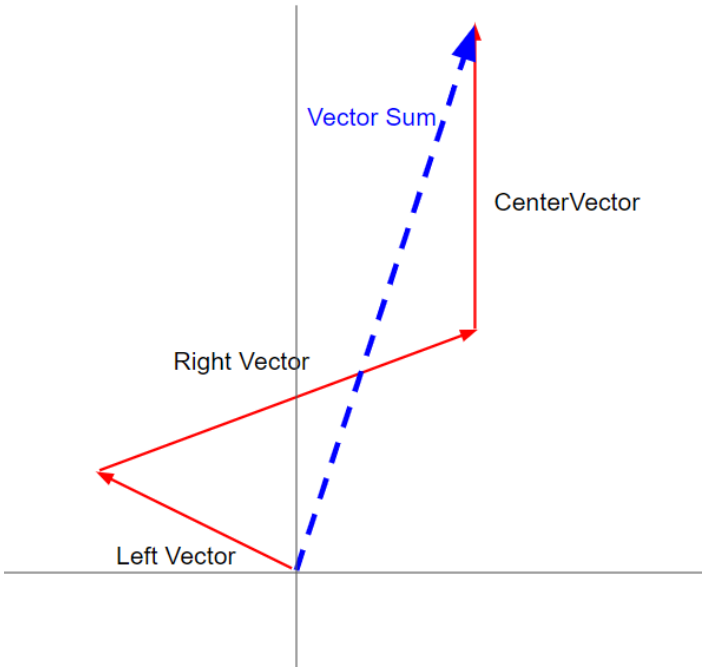

Figure 4 (above): Diagram of lidar distance vectors added together for direction from DIY Robocars. (2020, March 19). Lessons learned making a Lidar-based maze rover https://www.diyrobocars.com/2020/03/19/lessons-learned-making-a-lidar-based-maze-rover/

IV.B. Simulation

When simulating the Turtlebot in gazebo, we ran into a few issues:

1. Creating the package for maze_simulation was pretty easy but getting the naming conventions of the models to work with the launch files and world files gave us some issues. The URI for the models wasn't working when we opened up the gazebo simulation due to it being in the wrong folder. The setup.py file also gave us a little bit of trouble to call on the correct world and maze files.
2. The navigation code from the software side caused the Turtlebot to only turn in circles in the gazebo simulation. In the real world, the Turtlebot had to navigate through a maze of smaller proportions than in the simulation. If the front was clear in the simulation and the front left average was the largest value, the Turtlebot would just drive itself in circles. To fix this, we removed the part where it would consider the obstacle distance to the left and right if the front was open. We instead forced the robot to simply drive in a straight line until it met an obstacle. This worked a lot better in the simulation than it did in the real world due to the size of the mazes.

Work Division

Person	Work
Swetha Pallerla	- Wrote movement code - Tested on real-world turtlebot - Tested different speeds - Wrote introduction beginning, introduction hardware, hardware results sections
Olivia Adams	- Wrote movement code - Tested on real-world turtlebot - Tested different speeds - Wrote hardware methods and retrospect software section
Kevin Yen	- Wrote simulation launch and world files - Modified hardware code to work better for Gazebo simulation - Wrote introduction software, software methods and results sections
Kush Patel	- Got the gazebo simulation working with the hw5 package - Debugged simulation launch and world files - Wrote the retrospect simulation section

Code

Hardware

```
#Swetha Pallerla, Olivia Adams
#ENAE450, University of Maryland

import rclpy
import math
from rclpy.node import Node
import time

from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan


class Move(Node):

    def __init__(self):
        super().__init__('Move') # node name
        # init vars
        timer_period = 0.1  # seconds
        self.msg = Twist()
        self.time = 0
        self.lidar_data = LaserScan()

        # FRONT ANGLE CALIBRATION
        self.front = 360
        self.scan_range = 10*2 # x degrees * 2 = 2x units
        self.front_scan_min = self.front - self.scan_range
        self.front_scan_max = self.front + self.scan_range
        self.left_scan_min = self.front + 180 - self.scan_range
        self.left_scan_max = self.front + 180 + self.scan_range
        self.right_scan_min = self.front - 180 - self.scan_range
        self.right_scan_max = self.front - 180 + self.scan_range
        self.min_dist_from_obstacle = 0.35

        # PUBLISHER, SUBSCRIBER, TIMER, START MOVING
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10) # publisher, msgType = Twist, topic =
cmd_vel
        self.subscriber = self.create_subscription(LaserScan, 'scan', self.scan_callback, 10) # subscriber,
msgType = LaserScan, topic = scan, run scan_callback every time msg is received
        self.timer = self.create_timer(timer_period, self.timer_callback) # timer


    def timer_callback(self):
        self.time +=1


    # PURPOSE: set linear and angular speed of turtlebot
    def set_move(self, speed, angle):
        self.msg.linear.x = speed
        self.msg.angular.z = angle
        self.publisher_.publish(self.msg)


    # PURPOSE: turn desired number of degrees. left = +, right = -
    def turn(self, angle_degree):
        angle_rad = angle_degree * math.pi / 180
        current_time = self.time
        self.set_move(0.0, angle_rad)
        time.sleep(1)
        self.set_move(0.0, 0.0)
        time.sleep(1)


    # PURPOSE:  in range [min_angle, max_angle] of LaserScan msg, find index of minimum distance, and minimum
distance [m]
    # RETURNS:  index of minimum distance, and minimum distance [m]
    def get_min_distance_in_range(self, min_angle, max_angle, msg):
        min_distance = 100000
        min_index = -1
        for index,range in enumerate(msg.ranges[min_angle:max_angle]):
            if range > 0.0 and range != float('inf') and range < min_distance:
                min_index = index + min_angle # need to add min angle because we've chopped down the list.
                min_distance = range
```

```
        self.get_logger().info('Minimum distance of %s at index %s' % (min_distance, min_index))
        return min_index, min_distance


    # PURPOSE:  given a LaserScan object msg, find the average distance in the angular range given (within
0-720), filtering out large values
    # RETURNS:  average distance, minimum distance found in range
    def get_average_distance(self, min_angle, max_angle, msg):
        dist_in_range = []

        # filter out unreasonably large/inf distance values
        for index,range in enumerate(msg.ranges[min_angle:max_angle]):
            if range < 100: # filter to remove inf values
                dist_in_range.append(range)

        # return average and minimum of filtered values
        average = sum(dist_in_range)/len(dist_in_range)
        return average, min(dist_in_range)


    # PURPOSE:  basic algorithm for tight mazes (corridors approx 2x turtlebot width) with 90deg corners
    def simple_nav(self, msg):
        self.set_move(0.1, 0.0) # start moving
        min_index, min_distance = self.get_min_distance_in_range(self.front_scan_min, self.front_scan_max, msg)
# find the closest measurement and its index

        if min_distance < self.min_dist_from_obstacle: # if there is a small distance from obstacle
            self.get_logger().info("Close object distance: %s" % min_distance)

            self.set_move(0.0, 0.0) # stop moving

            min_distance_left = self.get_min_distance_in_range(self.left_scan_min, self.left_scan_max, msg)[1]
            min_distance_right = self.get_min_distance_in_range(self.right_scan_min, self.right_scan_max,
msg)[1]
            self.get_logger().info("distance to left %s" % min_distance_left)
            self.get_logger().info("distance to right %s" % min_distance_right)

            # turning logic
            if min_distance_left < min_distance_right:
                self.turn(-95) # after testing, 95 gives 90 degree turn
                self.get_logger().info("i'm going to turn right")
            else:
                self.turn(95)
                self.get_logger().info("i'm going to turn left")


    # PURPOSE:  navigate maze using vector summation method
    def vector_nav(self, msg):
        buffer_time = 0.25

        # find average distances to left, right, center
        left_dist = self.get_average_distance(360+40, 360+3*40, msg)
        right_dist = self.get_average_distance(360-3*40, 360-40, msg)
        center_dist = self.get_average_distance(360-40, 360+40, msg)
        self.get_logger().info("L dist: %s" % str(left_dist))
        self.get_logger().info("R dist: %s" % str(right_dist))
        self.get_logger().info("C dist: %s" % str(center_dist))

        # convert to vectors
        left_y = math.sin(math.pi/4) * left_dist
        left_x = -1*math.cos(math.pi/4) * left_dist
        right_y = math.sin(math.pi/4) * right_dist
        right_x = math.cos(math.pi/4) * right_dist
        center_y = center_dist/2 # halve the center vector because avoiding the walls to right and left is more
important than seeking some distant free space

        # add vectors together
        sum_x = round(left_x + right_x,2)
        sum_y = round(center_y - (left_y + right_y)/2,2)
        sum_angle = math.atan2(sum_x,sum_y)

        self.get_logger().info("x direction: %s" % str(sum_x))
        self.get_logger().info("y direction: %s" % str(sum_y))
        self.get_logger().info("sum ang: %s" % str(sum_angle))

        angular_velocity = sum_angle/buffer_time
        self.set_move(0.05, angular_velocity)
        time.sleep(buffer_time)
```

```
    # PURPOSE:  navigate maze using 3 bucket (left, center, right) method. evaluate each bucket
    def zone_nav(self, msg):
        # find average distance and minimum distance in each bucket
        left_dist, left_min = self.get_average_distance(460, 620, msg)
        right_dist, right_min = self.get_average_distance(100, 260, msg)
        center_dist, center_min = self.get_average_distance(280, 440, msg)
        averages = [left_dist, right_dist, center_dist]

        self.get_logger().info("L dist: %s, %s" % (str(left_dist), str(left_min)))
        self.get_logger().info("R dist: %s, %s" % (str(right_dist), str(right_min)))
        self.get_logger().info("C dist: %s, %s" % (str(center_dist), str(center_min)))


        # if there is no front collision risk
        if center_min > self.min_dist_from_obstacle:
            if max(averages) == left_dist: # if the left bucket has the most open space, go forward-left
                self.set_move(0.26, 1.0)
                self.get_logger().info("forward left")

            elif max(averages) == right_dist: # if the right bucket has the most open space, go forward-right
                self.set_move(0.26, -1.0)
                self.get_logger().info("forward right")

            elif max(averages) == center_dist: # if the center bucket has the most open space, go forward
                self.set_move(0.26, 0.0)
                self.get_logger().info("forward center")

        # if there is collision risk, 0 linear velocity
        else:
            if left_dist > right_dist: # if the left bucket has the most open space, turn towards it
                self.set_move(0.0, 1.0)
                self.get_logger().info("turn left")
            elif right_dist > left_dist: # if the right bucket has the most open space, turn towards it
                self.set_move(0.0, -1.0)
                self.get_logger().info("turn right")


    def scan_callback(self, msg):
        self.zone_nav(msg)


def main(args=None):
    rclpy.init(args=args)
    move_node = Move()

    rclpy.spin(move_node)

    move_node.destroy_node()# destroy the node explicitly
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

Simulation

Launch File - Maze 0

```
#!/usr/bin/env python3
#
# Copyright 2019 ROBOTIS CO., LTD.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

```
#
# Authors: Joep Tool

import os

from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def generate_launch_description():
    launch_file_dir = os.path.join(get_package_share_directory('turtlebot3_gazebo'), 'launch')
    pkg_gazebo_ros = get_package_share_directory('gazebo_ros')

    use_sim_time = LaunchConfiguration('use_sim_time', default='true')
    x_pose = LaunchConfiguration('x_pose', default='-0.0')
    y_pose = LaunchConfiguration('y_pose', default='-0.0')

    world = os.path.join(
        get_package_share_directory('maze_simulation'),
        'worlds',
        'maze_0.world'
    )

    gzserver_cmd = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(pkg_gazebo_ros, 'launch', 'gzserver.launch.py')
        ),
        launch_arguments={'world': world}.items()
    )
  gzclient_cmd = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(pkg_gazebo_ros, 'launch', 'gzclient.launch.py')
        )
    )

    robot_state_publisher_cmd = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(launch_file_dir, 'robot_state_publisher.launch.py')
        ),
        launch_arguments={'use_sim_time': use_sim_time}.items()
    )

    spawn_turtlebot_cmd = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(launch_file_dir, 'spawn_turtlebot3.launch.py')
        ),
        launch_arguments={
            'x_pose': x_pose,
            'y_pose': y_pose
        }.items()
    )

    navigator_node = Node(
        package="maze_simulation",
        executable="navigator"
    )

    ld = LaunchDescription()

    # Add the commands to the launch description
    ld.add_action(gzserver_cmd)
    ld.add_action(gzclient_cmd)
    ld.add_action(robot_state_publisher_cmd)
    ld.add_action(spawn_turtlebot_cmd)
    ld.add_action(navigator_node)
```

```
    return ld
```

World Files - Maze 0

```
<?xml version="1.0"?>
<sdf version="1.7">
  <world name="default">

    <include>
      <uri>model://ground_plane</uri>
    </include>

    <include>
      <uri>model://sun</uri>
    </include>

    <scene>
      <shadows>false</shadows>
    </scene>

    <gui fullscreen='0'>
      <camera name='user_camera'>
        <pose frame=''>0.319654 -0.235002 9.29441 0 1.5138 0.009599</pose>
        <view_controller>orbit</view_controller>
        <projection_type>perspective</projection_type>
      </camera>
    </gui>

    <physics type="ode">
      <real_time_update_rate>1000.0</real_time_update_rate>
      <max_step_size>0.001</max_step_size>
      <real_time_factor>1</real_time_factor>
      <ode>
        <solver>
          <type>quick</type>
          <iters>150</iters>
          <precon_iters>0</precon_iters>
          <sor>1.400000</sor>
          <use_dynamic_moi_rescaling>1</use_dynamic_moi_rescaling>
        </solver>
        <constraints>
          <cfm>0.00001</cfm>
          <erp>0.2</erp>
          <contact_max_correcting_vel>2000.000000</contact_max_correcting_vel>
          <contact_surface_layer>0.01000</contact_surface_layer>
        </constraints>
      </ode>
    </physics>

    <model name="maze_0">
      <static>1</static>
      <include>
        <uri>model://maze_0</uri>
      </include>
    </model>
  </world>
</sdf>
```

Simulation Navigation Code

```
import rclpy
import math
from rclpy.node import Node
import time

from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan


class Move(Node):

    def __init__(self):
        super().__init__('Move') # node name
```

```
        #init vars
        timer_period = 0.1  # seconds
        self.msg = Twist()
        self.time = 0
        self.lidar_data = LaserScan()
        self.complete = False
        # left side (say 20 to 100 degrees, if straight ahead is 0 degrees), a right side (260 to 340 degrees)
and a center (340 to 20 degrees).

        # FRONT ANGLE CALIBRATION
        self.front = 360 # CALIBRATE EACH TIME
        self.scan_range = 10*2 # x degrees * 2 = 2x units
        self.front_scan_min = self.front - self.scan_range
        self.front_scan_max = self.front + self.scan_range
        self.left_scan_min = self.front + 180 - self.scan_range
        self.left_scan_max = self.front + 180 + self.scan_range
        self.right_scan_min = self.front - 180 - self.scan_range
        self.right_scan_max = self.front - 180 + self.scan_range
        self.min_dist_from_obstacle = 0.34

        # PUBLISHER, SUBSCRIBER, TIMER, START MOVING
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10) # type twist to topic cmd_vel
        self.subscriber = self.create_subscription(LaserScan, 'scan', self.scan_callback, 10)
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.set_move(0.0, 0.0)


    def timer_callback(self):
        self.time +=1
        #self.vector_nav(self.lidar_data)
        """
        if self.time == 10:
            self.set_move(0.0, 0.0)
            self.get_logger().info('TIMER STOP')"""

    def set_move(self, speed, angle):
        self.msg.linear.x = speed
        self.msg.angular.z = angle

        self.publisher_.publish(self.msg)
       #self.get_logger().info('velocity: "%s"' % self.msg.linear.x)

    # left = 90, right = -90
    def turn(self, angle_degree):
        angle_rad = angle_degree * math.pi / 180
        current_time = self.time
        self.set_move(0.0, angle_rad)
        time.sleep(1)
        self.set_move(0.0, 0.0)
        time.sleep(1)

    def get_data(self, msg):
        dist = []
        for index, range in enumerate(msg.ranges[460:620]):
            self.get_logger().info('Distance: %s' % range)
            if range > 0 and range < 100:
                dist.append(range)
        return dist

    def get_min_distance_in_range(self, min_angle, max_angle, msg):
        min_distance = 100000
        min_index = -1
        for index,range in enumerate(msg.ranges[min_angle:max_angle]):
            if range > 0.0 and range != float('inf') and range < min_distance:
                min_index = index + min_angle # need to add min angle because we've chopped down the list. this
doenst make sense because wont the values in the tuple stay the same?
                min_distance = range

        self.get_logger().info('Minimum distance of %s at index %s' % (min_distance, min_index))
        return min_index, min_distance

    def nav(self, msg):
        self.set_move(0.1, 0.0)
        min_index, min_distance = self.get_min_distance_in_range(self.front_scan_min, self.front_scan_max, msg)
        if min_distance < self.min_dist_from_obstacle:
            self.get_logger().info("ruh roh object is %s" % min_distance)

            self.set_move(0.0, 0.0)
```

```
            min_distance_left = self.get_min_distance_in_range(self.left_scan_min, self.left_scan_max, msg)[1]
            min_distance_right = self.get_min_distance_in_range(self.right_scan_min, self.right_scan_max,
msg)[1]
            self.get_logger().info("distance to left %s" % min_distance_left)
            self.get_logger().info("distance to right %s" % min_distance_right)
            if min_distance_left < min_distance_right:
                self.turn(-95)
                self.get_logger().info("i'm going to turn right")
            else:
                self.turn(95)
                self.get_logger().info("i'm going to turn left")

    def get_average_distance(self, min_angle, max_angle, msg):
        dist_in_range = []
        for index,range in enumerate(msg.ranges[min_angle:max_angle]):
            if range > 10: # filter to remove inf values
                dist_in_range.append(10)
            else:
                dist_in_range.append(range)
        #self.get_logger().info("Ranges: %s" % str(dist_in_range))
        if len(dist_in_range) == 0:
            dist_in_range.append(100)
        average = sum(dist_in_range)/len(dist_in_range)
        return average, min(dist_in_range)

    def get_max_distance(self, min_angle, max_angle, msg):
        dist_in_range = []
        for index,range in enumerate(msg.ranges[min_angle:max_angle]):
            if range > 10: # filter to remove inf values
                dist_in_range.append(10)
            else:
                dist_in_range.append(range)
        #self.get_logger().info("Ranges: %s" % str(dist_in_range))
        if len(dist_in_range) == 0:
            dist_in_range.append(100)
        return max(dist_in_range)

    def vector_nav(self, msg):
        # get average distances
        # front is 360

        buffer_time = 0.25
        left_dist = self.get_average_distance(360+40, 360+3*40, msg)
        right_dist = self.get_average_distance(360-3*40, 360-40, msg)
        center_dist = self.get_average_distance(360-40, 360+40, msg)
        self.get_logger().info("L dist: %s" % str(left_dist))
        self.get_logger().info("R dist: %s" % str(right_dist))
        self.get_logger().info("C dist: %s" % str(center_dist))

        # convert to vectors
        left_y = math.sin(math.pi/4) * left_dist
        left_x = -1*math.cos(math.pi/4) * left_dist
        right_y = math.sin(math.pi/4) * right_dist
        right_x = math.cos(math.pi/4) * right_dist
        center_y = center_dist # halve the center vector because avoiding the walls to right and left is more
important than seeking some distant free space

        # add vectors together
        sum_x = round(left_x + right_x,2)
        sum_y = round(center_y - (left_y + right_y)/2,2)
        #if sum_y < 100:
        #    sum_y = 100
        sum_angle = math.atan2(sum_x,sum_y)

        self.get_logger().info("x direction: %s" % str(sum_x))
        self.get_logger().info("y direction: %s" % str(sum_y))
        self.get_logger().info("sum ang: %s" % str(sum_angle))

        # CHECK: is sum_angle in degrees or radians
        angular_velocity = sum_angle/buffer_time
        #angular_velocity = sum_angle
        self.set_move(0.05, angular_velocity)
        time.sleep(buffer_time)

    def zone_nav(self, msg):
        '''
        average distance in 3 buckets
        if all 3 averages > 2R: # if we're not gonna hit something
            if max(averages) is avL:
```

```
                set angVel = 0.1 # turn left + go straight
                set linVel = 0.1
            else if max(averages) is avR:
                set angVel = -0.1 # turn right + go straight
                set linVel = 0.1
            else if max(averages) is avC:
                set angVel = 0 # don't turn + go straight
                set linVel = 0.1

        else: # we are gonna run into something
            if max(averages) is avL:
                set angVel = 0.1 # turn left
                set linVel = 0
            else if max(averages) is avR:
                set angVel = -0.1 # turn right
                set linVel = 0
        '''
        #self.get_logger().info("Scan Ranges: %s" % msg.ranges)
        left_dist, left_min = self.get_average_distance(50, 130, msg)
        right_dist, right_min = self.get_average_distance(230, 310, msg)
        center_left_dist, center_left_min = self.get_average_distance(0, 30, msg)
        center_right_dist, center_right_min = self.get_average_distance(330, 360, msg)
        center_dist = (center_left_dist + center_right_dist)/2
        center_min = min(center_left_min, center_right_min)
        averages = [left_dist, right_dist, center_dist]

        self.get_logger().info("L dist: %s, %s" % (str(left_dist), str(left_min)))
        self.get_logger().info("R dist: %s, %s" % (str(right_dist), str(right_min)))
        self.get_logger().info("C dist: %s, %s" % (str(center_dist), str(center_min)))

        if self.complete == False:
            if left_dist >= 5 and right_dist >= 5 and center_dist >= 5:
                self.get_logger().info("Complete")
                self.set_move(0.0, 0.0)
                time.sleep(0.25)
                self.set_move(0.0, 1.0)
                time.sleep(2*3.14) #2pi seconds to account for Gazebo real time factor being about 0.5 for my
computer
                self.set_move(0.0, 0.0)
                self.complete = True
            # if we're not gonna hit something
            elif center_min > self.min_dist_from_obstacle:
                self.get_logger().info("in if")
                '''
                if max(averages) == left_dist:
                    self.set_move(0.26, 1.0)
                    self.get_logger().info("forward left")
                elif max(averages) == right_dist:
                    self.set_move(0.26, -1.0)
                    self.get_logger().info("forward right")
                elif max(averages) == center_dist:
                    self.set_move(0.26, 0.0)
                    self.get_logger().info("forward center")
                '''
                self.set_move(0.26, 0.0)

            # if we're gonna hit something, 0 linear, just turn
            else:
                self.get_logger().info("in else")
                if center_left_dist > 0.35:
                    self.set_move(0.0, 1.0)
                    self.get_logger().info("turn center left")
                elif center_right_dist > 0.35:
                    self.set_move(0.0, -1.0)
                    self.get_logger().info("turn center right")
                if left_dist - right_dist > 0.15:
                    self.set_move(0.0, 1.0)
                    self.get_logger().info("turn left")
                elif right_dist - left_dist > 0.15:
                    self.set_move(0.0, -1.0)
                    self.get_logger().info("turn right")
                else:
                    self.set_move(-0.26, -0.5)
                    time.sleep(0.5)
                    self.get_logger().info("back up")


            '''
            if abs(left_dist - right_dist) < 0.15:
```

```
                time.sleep(3)
                self.get_logger().info("convex corner pause")
            '''

    #write code to avoid in here
    def scan_callback(self, msg):
        #self.nav(msg)
        #self.lidar_data.ranges = msg.ranges
        #self.vector_nav(msg)
        #if len(self.get_data(msg)) > 0:
            #self.get_logger().info("data array length: " + str(self.get_data(msg)))
        self.zone_nav(msg)
        #self.set_move(0.1, 0.0)
        '''
        min_index, min_distance = self.get_min_distance_in_range(self.front_scan_min, self.front_scan_max, msg)

        if min_distance < self.min_dist_from_obstacle:
          #  self.set_move(0.0, 0.0)
            self.get_logger().info('E STOP')'''

        # INDEX REFERENCE
        # forward = 364
        # right = 180
        # left = 520
        # behind = 0
        # scanner increments by 0.5 degrees (0-45 degrees -- > 0-90 list indexes)

    """def __del__(self):
        self.set_move(0.0, 0.0)
        super().__del__()
        self.get_logger().info('Node destroyed')"""


def main(args=None):
    rclpy.init(args=args)

    move_node = Move()

    rclpy.spin(move_node)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    move_node.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

Links

https://www.diyrobocars.com/2020/03/19/lessons-learned-making-a-lidar-based-maze-rover/