Autonomous obstacle avoidance in Husarion ROSbot

By

Aseem Saxena

Department of Electrical and Electronics Engineering

University of Bath

Table of Contents

Executive Summary

Autonomous Driving could revolutionise the way consumers experience mobility. AD systems may make driving safer, more convenient, and more enjoyable. Hours on the road previously spent driving could be used to video call a friend, watch a funny movie, or even work. For employees with long commutes, driving an AV might increase worker productivity and even shorten the workday. Since workers can perform their jobs from an autonomous car, they could more easily move farther away from the office, which, in turn, could attract more people to rural areas and suburbs. (*The future of autonomous vehicles (AV) | McKinsey*, n.d.).

Facilitating these advancements calls for improved algorithms and novel architecture in both hardware and software. A fully functional Autonomous obstacle avoidance system running in parallel with other sensors in the mobile unit is the answer to this problem. This also provides huge business and infrastructure opportunities that can save billions of dollars in savings and save lives.

According to McKinsey, by 2035, autonomous driving could create $300 billion to $400 billion in revenue. New research reveals what's needed to win in the fast-changing passenger car market. (*The future of autonomous vehicles (AV) | McKinsey*, n.d.)

This project explores the possibility of building an Obstacle collision avoidance system to autonomously guide and move a Husarion ROSbot (4-wheeled mobile bot) to a specified location while autonomously avoiding obstacles using the robot's sensor data and the obstacle avoidance algorithm.

This report has been created to document the process of the design and development of an Autonomous obstacle avoidance system using Husarion ROSbot running on ROS (Robot Operating System) while explaining the algorithm, robot, software architecture, code implementation and simulating the robot in Gazebo to run test approaches and find limitations and further work to discuss potential future solutions.

The final product is a system which can be used in many applications in industrial robotics, for prototyping and developing land, aerial, and water-based mobile robots, and helping to build and develop systems to aid autonomous robotics with a hands-on.

Introduction

The project uses a Husarion ROSbot 2.0 robot. It is an autonomous robot platform based on Husarion CORE2-ROS robot controller, integrating, 4-wheels mobile platform containing DC motors with encoders and an aluminum frame, an Orbbec Astra RGBD camera attached beside a RPLIDAR A2 laser scanner, a MPU 9250 inertial sensor (accelerometer + gyro) and a Wi-Fi antenna. The robot also has four Time-of-Flight infrared distance sensors to sense the presence of an object within the range of up to 200 cm. Maximum translational velocity is 1.25 m/s.(*HUSARION ROSBOT SERIES MANUAL Pdf Download*, n.d.)

The robot comes with an inbuilt panel of GPIO and USB interface to establish communication with other devices. The microcontroller used to control the bot is an ASUS Tinker Board with 2GB memory and a Cortex A17 Quadcore 1.8 GHz Processor. The software used to control the ROSBot are Ubuntu Linux 20.04 LTS and ROS Noetic. ROSbot uses ROS software framework which has message-based architecture, and powerful debugging tools to aid the feature of software reuse.

The obstacles are compiled of multiple objects like tables, shelves, dustbins, walls and a human figure. The ROSbot needs to figure out the positioning of the obstacles that may have different shapes or structures and scale the path from a specific start point to a given end position in less time while calculating the distance covered by the robot.

We find a fitting algorithm for the robot and code it in Python to make the robot able to sense an object using a LIDAR Sensor and then get back to the correct path without colliding with any obstacles. There are multiple algorithms out there therefore, one needs to consider the execution of the program within the stipulated time and complexity.

Analysis of Design

The problem statement is about the design of an Autonomous Obstacle avoidance system. To simulate the environment, ROS with its dependencies and tools like Gazabo, rqt graph and Rviz are used. First, the breakdown of steps of operation.

- A goal is created in the workspace with quite a distance from the point of spawn of robot.
- The simulation world is then created with multiple objects like tables, bookshelves, walls, dustbins etc., and are placed in between the direct line of sight of starting point to the goal.
- The search of fitting algorithm for obstacle avoidance.
- Selection of sensors to use for creating odometry data for mapping and navigation purposes.
- Selection of a programming language to code the algorithm.
- Implementing the script within ROS architecture and screening the robot's behavior.
- Setting up test approaches for the algorithm and documenting the results.

The simulation world was created in Gazebo and the goal and point of spawn of robot along with obstacles are set.

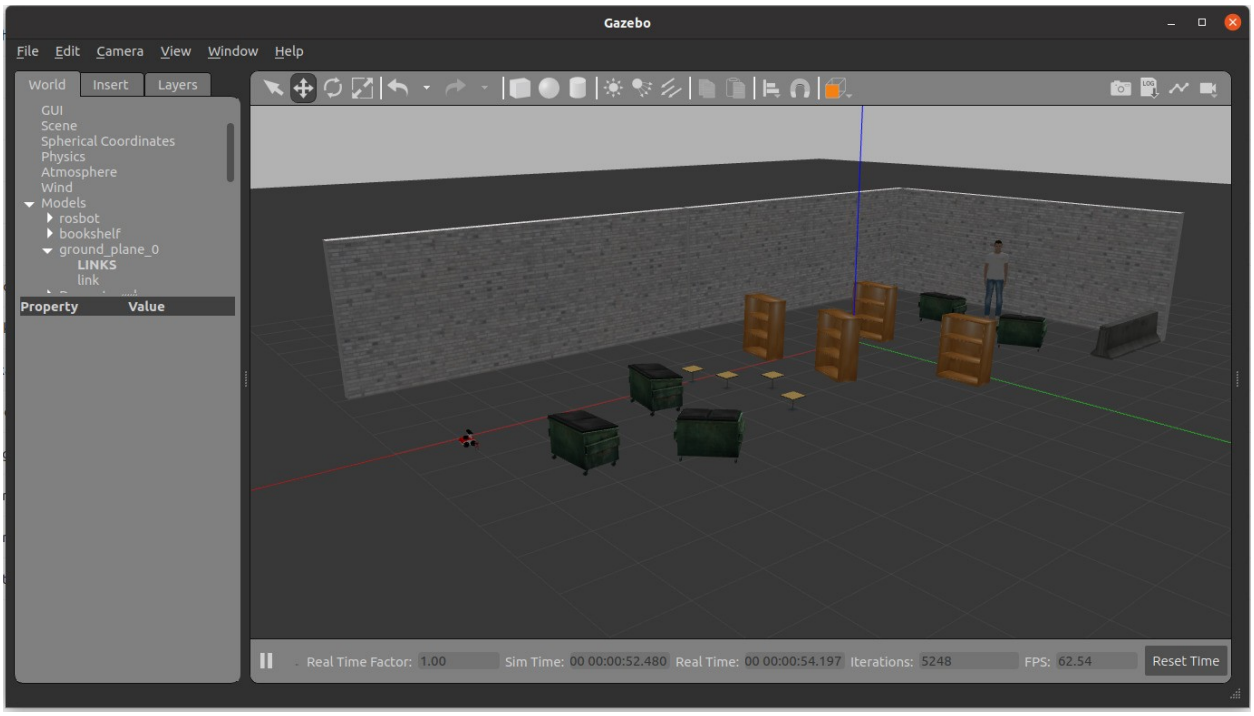


Fig. 1 gazebo simulation view of the world

Multiple algorithms for obstacle avoidance are available but the selection of the one fitting the best in the use case is crucial as it directly affects the complexity, time, and use of resources.

Similarly, the Husarion ROSbot have multiple sensors like Time-of-flight infrared sensor, RPLiDAR, and Orbbec Astra RGB camera with Depth perception, these are onboarding the platform, and an array of those sensors can be used to create a Senor fusion.

Sensor fusion defines as the process of combining sensor data or data derived from disparate sources such that the resulting information has less uncertainty. (Sensor fusion, 2023)

A digital camera is also an option which could be used for object detection, but the intensity of light might change from environment to environment as well in turn giving us different values. Although there are multiple ways and methods to incorporate sensor data, this project used the onboard RPLiDAR for detection and mapping of the obstacles in the workspace, as the Laser scan is more accurate and time efficient when mapping a bigger area with high precision. Also, its nature of mapping the data points continuously with time while rotating 360 degrees creates more reliable dataset as any changes in the environment can be mapped.

The LiDAR scanner can perform a scan within the maximum range of 12m at a frequency of 10Hz(600 rpm) which could be modulated between 5-15 Hz as per the requirement. For this specific simulation, the range for the LIDAR is taken to be 5m after running tests on the bot with consideration to all obstacles being used during the simulation phase. The robot needs to analyze the local design of the path to get a better hold of the turns. Since the robot already has inbuilt sensors like an IMU Sensor, an Infrared distance sensor, a digital camera and a LIDAR sensor the task of analyzing the path structure becomes much easier.

LIDAR will be used for the generation of 2D cloud data points to map the obstacles on the path and localize them for further developments in the program. The basic working of LIDAR is that it rotates in a clockwise direction to perform a 360° scan and emit a modulated infrared signal and once that signal is reflected back from any object present within the range of 3m then the signal is sampled through the Vision Acquisition System of RP LIDAR.

Synthesis of Design

For synthesizing the data generated through the LIDAR and to autonomously move while avoiding obstacles the ROSBot needs to follow an algorithm. So to complete the task of avoiding an obstacle can be done locally, globally or through having a combination of both. According to (Choset, 2005) Algorithms such as Bug 1, Bug 2 and Tangent Bug could be used to avoid obstacles locally without the need to construct a configuration space.

Bug 2 Algorithm

According to (Van Breda, 2016), the Bug 2 algorithm is an approach where the robot will not circumnavigate the robot for the determination of a leaving point instead it will calculate the shortest line that connects the robot's start and goal positions.

For instance, (Van Breda, 2016)explained the Bug 2 algorithm which is generated in Fig 2. In this figure, the algorithm calculated the shortest line or the m-line between the start and goal positions pretty easily. The reason for that is the m-line only has one obstacle at a time.

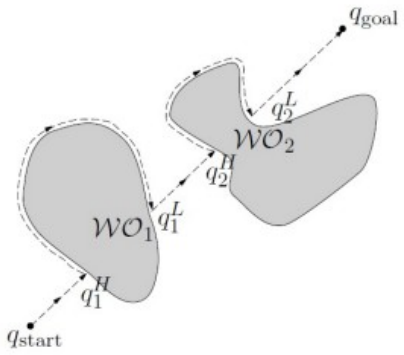


Fig 2. Bug 2 Algorithm

(Choset, 2005) described the Bug 2 algorithm as an opportunistic approach since it takes the first valid leave point available on the m-line and continues until it is back on the m-line. Despite being an algorithm developing a short line path for simpler obstacles there are still better obstacle avoidance algorithms to tackle the problem.

Tangent Bug Algorithm

To improvise the shortcomings of the Bug 2 algorithm using the m-line, we use the Tangent bug algorithm with the help of LIDAR to generate a path from start to goal.

The Tangent Bug algorithm's motion-to-goal and boundary-following behaviors differ from the Bug-2 implementations. The motion-to-goal behaviour, in this case, directs the robot towards the goal but might have a phase where the robot follows the boundary, while the boundary-following behaviour may have a phase where the robot does not follow the obstacle boundary (Howie Choset et al., 2005).

The Tangent Bug algorithm is affected by the range of its sensors. Two cases of the Tangent Bug algorithm are used to illustrate the effect of sensor range on the algorithm. In the first case the robot uses contact sensors (zero sensor range) and in the second case the robot's sensors have infinite range (Van Breda 2016).

The first case where only contact sensors are used is illustrated in Figure 3. As per Howie Choset et al. (2005), the robot in motion-to-goal behavior, moves towards the goal until it encounters the first obstacle at hit-point H1, from where it moves around the obstacle and departs from the obstacle boundary at depart-point D1.

Next, the robot moves towards the goal until it encounters the second obstacle at H2, from where it continues to move around the obstacle boundary in motion to-goal behaviour to minimize the heuristic distance of the robot to the goal.

This process continues until the robot eventually reaches a local minimum at M3 where the robot is unable to minimize the heuristic distance anymore.

At this point, the robot switches to the boundary-following behaviour until it reaches the leave-point at L3 where dreach < dfollowed is satisfied. Subsequently, the robot moves towards qgoal until it encounters another obstacle at hit-point H4.

The robot continues in motion-to-goal behaviour until it reaches a local minimum at M4 from where it switches to boundary-following behaviour. Lastly, the robot reaches leave-point L4, where dreach < dfollowed is satisfied, and continues to qgoal (Howie Choset et al., 2005).

Motion-to-goal behaviour is indicated with dotted lines while boundary-following behaviour is indicated with dashed lines (Howie Choset et al., 2005).

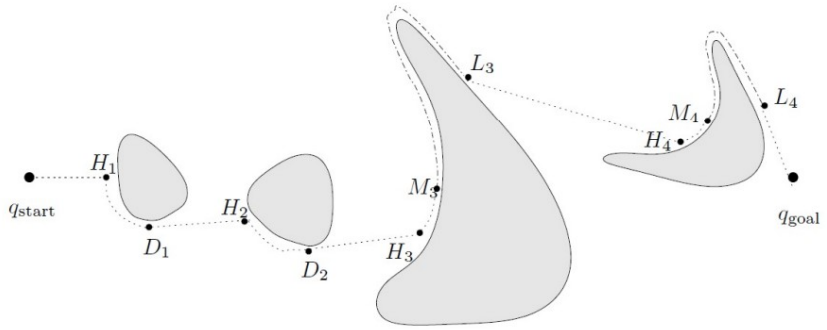


Fig 3. Tangent Bug example with zero sensor range

In the second case according to Howie Choset et al.(2005), illustrated in Figure 2.9, the robot's infinite sensor range allows it to react immediately to the first obstacle and thus qstart and H1 coincide.

The robot moves around the obstacle and departs from it at departpoint D1 which also coincides with hit-point H2. Then in the same manner as before the robot moves around the second obstacle and departs from it at D2, which coincides with hit-point H3.

Next, the robot can move around the third obstacle without falling into a local minimum as was the case when contact sensors were used. Finally, the robot departs from the third obstacle at D3 and continues to the goal (Howie Choset et al., 2005).

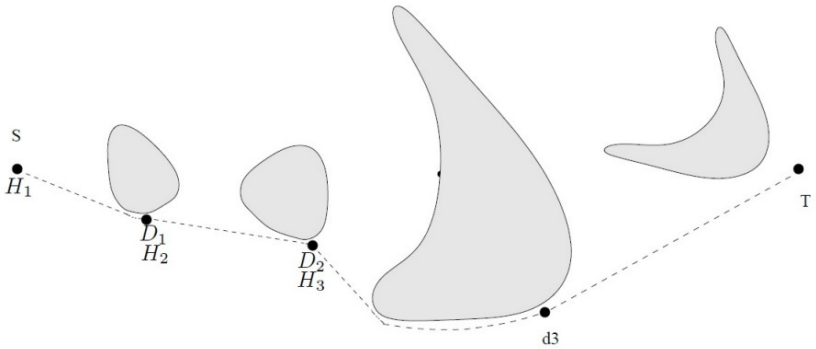


Fig 4. Tangent Bug algorithm with infinite sensor range

Software Architecture

In Fig. 5. The ROS rqt graph represents the communication between different nodes within a multi-robot simulation environment. At the heart of the graph is the /stage node, which likely corresponds to the Stage simulator—a 2D robot simulator compatible with ROS. The Stage simulator provides a virtual environment where robot models can move and interact.

Surrounding the /stage node are nodes representing three individual robots, as indicated by the prefixes /robot_0, /robot_1, and /robot_2. Each of these robot nodes has associated topics for commanding and monitoring the robot's state within the simulation:

- */robot_X/cmd_vel* are topics where velocity commands (cmd_vel stands for 'command velocity') are published to control the linear and angular velocity of each robot (where X represents the robot's index, 0, 1, or 2).

- /robot_X/odom are topics for odometry information, which detail the robot's position and orientation based on motion sensors and wheel encoder data.

- /robot_X/base_scan are likely topics for laser scan data, which contain information about the proximity of obstacles around each robot. This data is instrumental for navigation and obstacle avoidance.

- /robot_X/base_pose_ground_truth topics probably provide the actual position and orientation of each robot within the simulation. This 'ground truth' data is typically used for comparison with estimated positions to evaluate the accuracy of localization algorithms.

The /tf topic is a transformation tree that keeps track of all coordinate frames relative to each other. This is crucial in multi-robot systems to understand the relative positions and orientations of different robots and objects in the environment.

The /clock topic likely distributes the simulation time to all nodes, ensuring synchronization across the simulated processes.

The /statistics topic might be publishing performance or diagnostic information about the communication network within the ROS graph, which can be used for analysis and debugging purposes. The nodes /rosout, /rosout_agg, and the /rqt_gui_py_node_28146 are part of ROS's logging and debugging infrastructure. /rosout is a consolidated log for messages from various nodes. /rosout_agg aggregates these messages, and the rqt_gui node represents a graphical user interface tool used for ROS diagnostics and introspection.

Each robot is independently receiving velocity commands, producing odometry and laser scan data, and has its own ground truth pose information. The central Stage node orchestrates the simulation, while the logging infrastructure collects and displays runtime information and errors. This setup is typical of a robotic research or development environment where multiple robots are being simulated to test algorithms for localization, mapping, navigation, and multi-robot coordination.

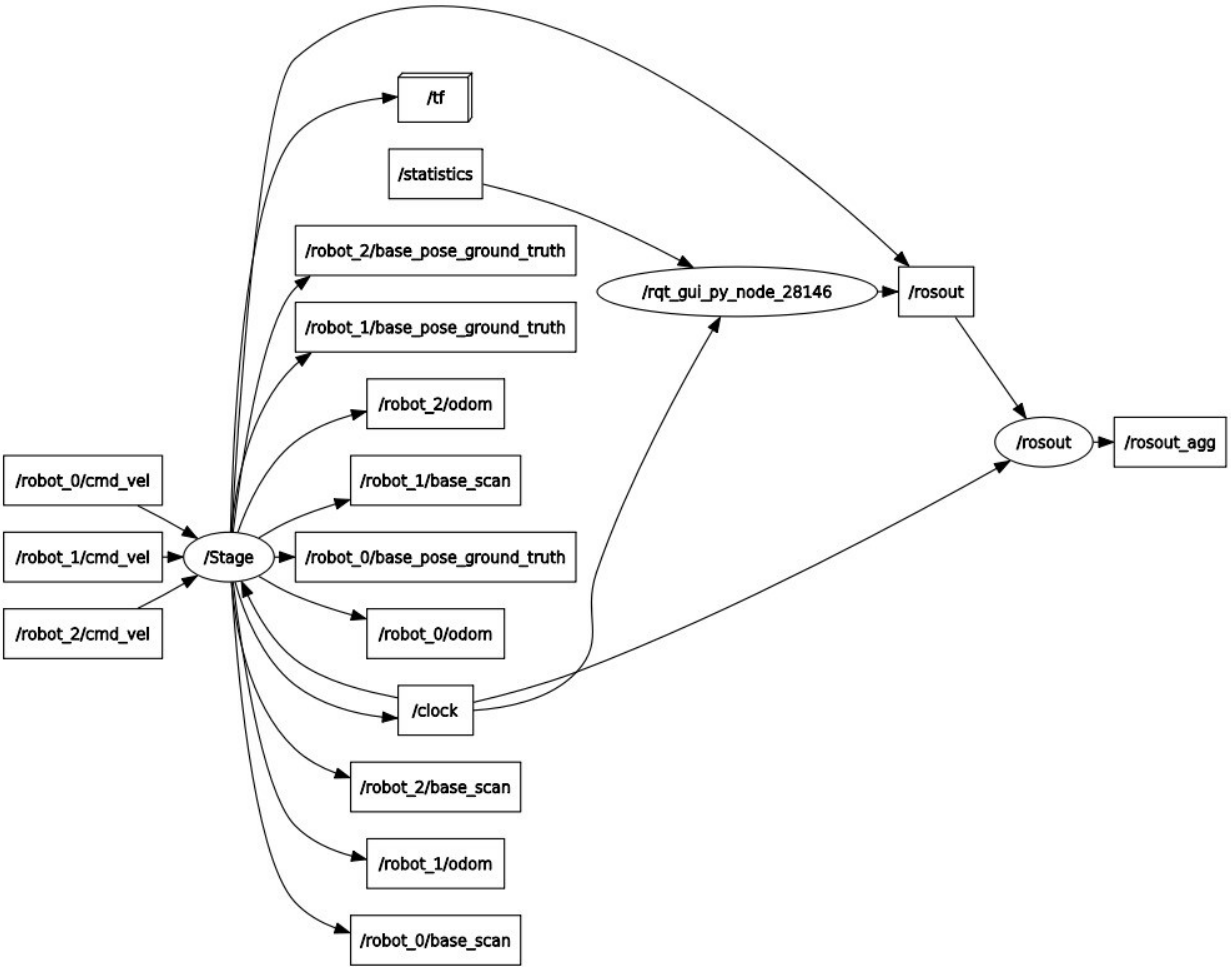


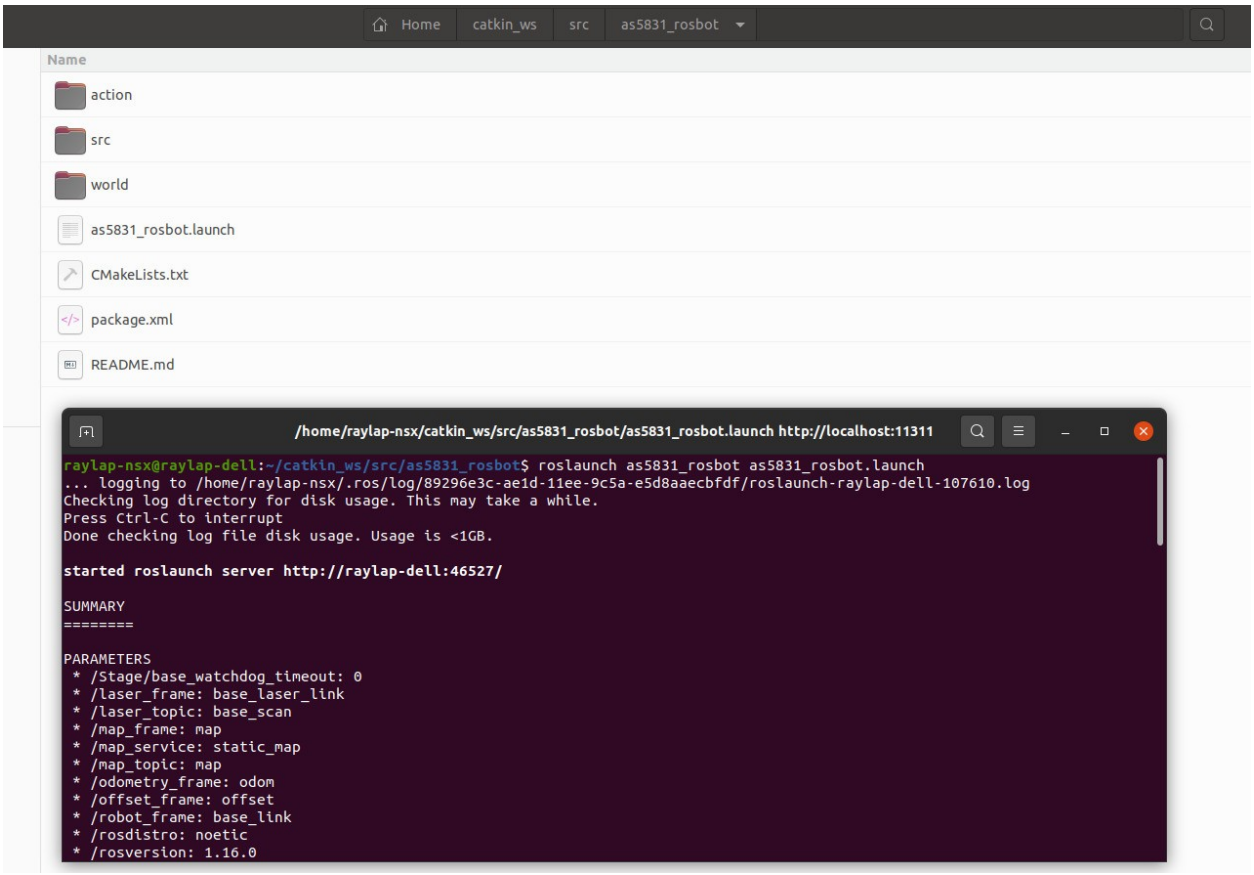
Fig. 5 rqt_graph for the software

Software Implementation

(*find all the code in appendices)

The code script is written in python and the launch file can be run by using command:

roslaunch as5831_rosbot as5831_rosbot.launch



This command calls forth the *as5831_rosbot.launch* file.

```
    <!-- Start Stage simulator with a given environment -->
    <node name="Stage" pkg="stage_ros" type="stageros" args="$(find as5831_rosbot)/world/as5831_rosbot.world">
        <param name="base_watchdog_timeout" value="0" />
    </node>

    <node name="robotframe0" pkg="as5831_rosbot" type="frames0.py"/>

    <node name="tangentbug0" pkg="as5831_rosbot" type="tangentbug0.py"/>

    <node name="actionServer" pkg="as5831_rosbot" type="actionServer.py"/>

    <node name="goal0" pkg="as5831_rosbot" type="goal0.py"/>
</launch>
```

And opens several nodes for frame.py, tangentbug.py, actionServer.py and goal.py.

Frame represents the frame of reference of the robot for measuring distance and time with respect to the simulation and keep track of the position of robot in the workspace.

frames0.py

```
#!/usr/bin/env python

import rospy
import tf
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist

# creates a fixed world frame at the coordinate origin called "world" in tf
# and creates robot0's odometry with frame
def worldframe(odom):
    br = tf.TransformBroadcaster()
    x = odom.pose.pose.position.x
    y = odom.pose.pose.position.y
    z = odom.pose.pose.position.z
    qx = odom.pose.pose.orientation.x
    qy = odom.pose.pose.orientation.y
    qz = odom.pose.pose.orientation.z
    qw = odom.pose.pose.orientation.w
    br.sendTransform((x,y,z),(qx,qy,qz,qw),rospy.Time.now(),"robot0/trueOdom","robot0/world")
```

The program imports dependencies rospy, tf, Odometry and Twist.

Rospy is a Python client library for ROS and enables Python language to quickly interface with ROS.

Tf is a package from *geometry* repo and enables the user to keep track of multiple coordinate frames over time. It also maintains the relationship between coordinate frames and time, and lets the user transform points, vectors between any two coordinate frames at any desired point in time.

Odometry is a type of message from navigation messages package which represents an estimate of a position and velocity in free space

Twist message is from the package *geometry_msgs* and expresses velocity in free space broken into its linear and angular parts.

As described in the code comments, This creates a fixed world frame at coordinate origin and creates robot's odometry with frame.

Defining *goalframe* creates a variable used to create the fixed goal frame.

```
# creates a fixed goal frame in tf
def goalframe(twi):
    br = tf.TransformBroadcaster()
    rate = rospy.Rate(10.0)
    x = twi.linear.x
    y = twi.linear.y
    br.sendTransform((x,y,0.0),(0.0,0.0,0.0,1.0),rospy.Time.now(),"robot0/goal","robot0/world")
```

```
if __name__ == '__main__':
    try:
        rospy.init_node('tf_boadcaster0',anonymous=True)
        rospy.Subscriber("/robot_0/base_pose_ground_truth",Odometry,worldframe)
        rospy.Subscriber("/robot_0/goal",Twist,goalframe)
        rospy.spin()
    except rospy.ROSInterruptException: pass
```

Actionserver provides actions, topics and services like robot position, goal server, simple action server and feedback running in our case. An action server has a name and a type. The name is allowed to be name spaced and must be unique across action servers.

actionServer.py

```
#! /usr/bin/env python

import rospy
import math
from geometry_msgs.msg import Twist
from geometry_msgs.msg import Vector3
from std_msgs.msg import Float64
from nav_msgs.msg import Odometry
import roslib
import actionlib

import as5831_rosbot.msg

robot0pos = Odometry();
robot1pos = Odometry();
robot2pos = Odometry();

def robot_0positionCheck(updatedState):
    global robot0pos
    robot0pos = updatedState
```

goal0.py determines the position of goal and communicates with *roslib* and *actionlib* to keep updating the goal position with reference to the odometry of the robot. The current goal is (x,y : 3,12)

goal0.py

```
#!/usr/bin/env python

import rospy
import tf
from geometry_msgs.msg import Twist
from geometry_msgs.msg import Vector3
from std_msgs.msg import Float64
import roslib
import actionlib

import as5831_rosbot.msg

currentGoal = Vector3(3.0,12.0,0.0)

def goal0_client():
    global currentGoal

    client=actionlib.SimpleActionClient('check_goals',as5831_rosbot.msg.goalStatusAction)
    client.wait_for_server()
    goal = as5831_rosbot.msg.goalStatusGoal(x=currentGoal.x,y=currentGoal.y)
    client.send_goal(goal)
    client.wait_for_result()
    return client.get_result()
```

This publishes a fixed goal frame by publishing *goalPub* and keeping the linear and angular position vector as zero.

goal0.py

```
25    # creates a fixed goal frame in tf
26    def goalPub():
27        global currentGoal
28        goalCount = 0
29
30        goalinworld = rospy.Publisher('robot_0/goal',Twist,queue_size=10)
31        goal = Twist()
32        goal.linear = Vector3(currentGoal.x,currentGoal.y,0.0)
33        goal.angular = Vector3(0.0,0.0,0.0)
```

rospy.Rate publishes the data by a fixed rate, for example, by 10 Hz in this case. make sure that the total execution time is lower than the frequency you give to the ROS Rate

goal0.py

```
34
35        rate = rospy.Rate(10.0)
36        while not rospy.is_shutdown():
37            result = goal0_client()
38            if result.robot0_thereOrNot == 1:
39                goalCount = goalCount + 1
40
41            if goalCount>=1:
42                currentGoal.x = 0.0
43                currentGoal.y = 0.0
44            else:
45                currentGoal.x = 3.0
46                currentGoal.y = 12.0
47
```

Testclient.py sets up a goal client and the client checks for the goal status action, it waits for the server to respond then send the fixed goal value (3,12).

testclient.py

```
1   #! /usr/bin/env python
2
3   import rospy
4   import tf
5   from geometry_msgs.msg import Twist
6   from geometry_msgs.msg import Vector3
7   from std_msgs.msg import Float64
8   import roslib
9   import actionlib
10
11  import as5831_rosbot.msg
12
13  def goal0_client():
14      client=actionlib.SimpleActionClient('robot0_check_goals',as5831_rosbot.msg.goalStatusAction)
15      client.wait_for_server()
16      goal = as5831_rosbot.msg.goalStatusGoal(x=3,y=12)
17      client.send_goal(goal)
18      client.wait_for_result()
19      return client.get_result()
```

Tangentbug0 is the main python code for the robot. The scripts call rospy, tf and math library. From different msg packages gets *Odometry, Twist, Laserscan, Vector3, UInt32.* The max range is set to 5 meters as at the greater distances, the lidar scan values brings a lot of noises.

tangentbug0.py

```
#!/usr/bin/env python

import rospy
import tf
import math
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Vector3
from std_msgs.msg import UInt32

maxRange = 5.0

goalVisible = 0

discThresh = 5.0

scan = LaserScan()

def scanUpdate(lScan):
    global scan
    scan = lScan

def tangentbug():
    global goalVisible
    global scan
    goalfromlaser = rospy.Publisher('robot_0/robot2goal',Twist,queue_size=10)
    robotController = rospy.Publisher('robot_0/cmd_vel',Twist,queue_size=10)
```

goalVisible determines if the robot can see the goal directly.

discthreshold defines if a point in the laser scan is a discontinuity.

scanUpdate updates the scans. *tangentbug*

```
def tangentbug():
    global goalVisible
    global scan
    goalfromlaser = rospy.Publisher('robot_0/robot2goal',Twist,queue_size=10)
    robotController = rospy.Publisher('robot_0/cmd_vel',Twist,queue_size=10)

    listener = tf.TransformListener()
    rate = rospy.Rate(1.0)
    while not rospy.is_shutdown():
        try:
            (trans,rot) = listener.lookupTransform('robot0/trueOdom','robot0/goal',rospy.Time(0))
            twi = Twist()
            twi.linear.x = trans[0]
            twi.linear.y = trans[1]
            twi.linear.z = trans[2]
            twi.angular = Vector3(0.0,0.0,0.0)
```

trans is a list of 3 elements x,y,z of the transform. *rot* is a list of 4 elements of the quaternion for the transform. And ignore angular orientations (*twi.angular*) since we only care about getting to the goal position.

This checks if we can see the goal directly and Move towards it if we can.

```
if (scan.ranges[sensorIndex] >= maxRange):
    goalVisible = 1
    if (angle2goal > 0):
        control.linear.x = 0.3
        control.angular.z = 0.2
    elif (angle2goal < 0):
        control.linear.x = 0.3
        control.angular.z = -0.2
    else:
        control.linear.x = 0.3
        control.angular.z = 0.0
else:
    goalVisible = 0
```

If we can't see the goal directly, then checks for the best direction of travel.

```
if goalVisible == 0:
    bestAngle = 0.0
    besti = 0
    bestDist = 10000.0
    realAngle = 0.0
```

This makes the robot drive towards the best heuristic or turn towards it if we're not facing it already.

```
if ((bestAngle) > 0):
    control.linear.x = 0.2
    control.angular.z = 0.5
elif ((bestAngle) < 0):
    control.linear.x = 0.2
    control.angular.z = -0.5
else:
    control.linear.x = 0.2
    control.angular.z = 0.0
```

if obstacles are too close to the robot, then prioritize avoiding them.

```
j = int(len(scan.ranges)/2) - 70
m = int(len(scan.ranges)/2) - 10
k = int(len(scan.ranges)/2) + 70
n = int(len(scan.ranges)/2) + 10

for i in range(j,m):
    if (scan.ranges[i] < 2.5):
        control.linear.x = 0.0
        control.angular.z = 0.5
for i in range(n,k):
    if (scan.ranges[i] < 2.5):
        control.linear.x = 0.0
        control.angular.z = -0.5
```

prioritize avoiding obstacles.

```
if (besti > 90) and (besti < (len(scan.ranges)-90)):
    if scan.ranges[besti+20] < 2.0:
        control.linear.x = 0.2
        control.angular.z = 0.5
    elif scan.ranges[besti-20] < 2.0:
        control.linear.x = 0.2
        control.angular.z = -0.5
elif (besti > 90) and (besti < (len(scan.ranges)-90)):
    if scan.ranges[besti+30] < 2.0:
        control.linear.x = 0.2
        control.angular.z = 0.5
    elif scan.ranges[besti-30] < 2.0:
        control.linear.x = 0.2
        control.angular.z = -0.5
```

stop moving if we're close enough to the goal.

```
    if math.sqrt((twi.linear.x) ** 2 + (twi.linear.y) ** 2) < 0.5:
        control.linear.x = 0.0
        control.linear.y = 0.0

    robotController.publish(control)
    goalfromlaser.publish(twi)
except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):
    continue
```

Testing Approach

Case 1

In this case, both robot and the goal are located on the same line. The planner initiates the motion to goal planner at first and switches to boundary following the obstacle is sensed.

Case 2

In this scenario, the robot faces a non-convex obstacle.

Case 3

This scenario is similar to case 2 apart from an added new obstacle that comes after the first one. The reason for creating the second obstacle was to test the repeatability of the planner algorithm.

Case 4

Differing from the rest of the cases, this case is a scenario in which the goal is within the obstacle regions.

Test Results

Case 1

The robot switched to boundary following the obstacle.

Case 2

The algorithm struggled to define the optimal approach to circumvent the obstacle.

Case 3

The repeatability of the planner algorithm was observed.

Case 4

Robot clings onto the boundary of obstacle and jitters.

References

1. Choset, H.M., ed., 2005. *Principles of robot motion: theory, algorithms, and implementation*. Intelligent robotics and autonomous agents. Cambridge, Mass: MIT Press.

2. *Documentation - ROS Wiki* [Online], n.d. Available from: http://wiki.ros.org/Documentation [Accessed 7 January 2024].

3. *The future of autonomous vehicles (AV) | McKinsey* [Online], n.d. Available from: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/autonomous-drivings-future-convenient-and-connected [Accessed 7 January 2024].

4. Van Breda, R., 2016. *Vector field histogram star obstacle avoidance system for multicopters* [Online]. Stellenbosch : Stellenbosch University. Available from: http://hdl.handle.net/10019.1/100319 [Accessed 7 January 2024].

Appendices

All the code for the project.

as5831_rosbot.launch

```
<launch>

    <!-- Some general parameters -->
    <param name="use_sim_time" value="true" />
    <rosparam file="$(find nav2d_tutorials)/param/ros.yaml"/>

    <!-- Start Stage simulator with a given environment -->
    <node name="Stage" pkg="stage_ros" type="stageros" args="$(find as5831_rosbot)/world/as5831_rosbot.world">
        <param name="base_watchdog_timeout" value="0" />
    </node>

    <node name="robotframe0" pkg="as5831_rosbot" type="frames0.py"/>

    <node name="tangentbug0" pkg="as5831_rosbot" type="tangentbug0.py"/>

    <node name="actionServer" pkg="as5831_rosbot" type="actionServer.py"/>

    <node name="goal0" pkg="as5831_rosbot" type="goal0.py"/>
</launch>
```

frames.py

```python
#!/usr/bin/env python

import rospy
import tf
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist

# creates a fixed world frame at the coordinate origin called "world" in tf
# and creates robot0's odometry with frame
def worldframe(odom):
    br = tf.TransformBroadcaster()
    x = odom.pose.pose.position.x
    y = odom.pose.pose.position.y
    z = odom.pose.pose.position.z
    qx = odom.pose.pose.orientation.x
    qy = odom.pose.pose.orientation.y
    qz = odom.pose.pose.orientation.z
    qw = odom.pose.pose.orientation.w
    br.sendTransform((x,y,z),(qx,qy,qz,qw),rospy.Time.now(),"robot0/trueOdom","robot0/world")

# creates a fixed goal frame in tf
def goalframe(twi):
    br = tf.TransformBroadcaster()
    rate = rospy.Rate(10.0)
    x = twi.linear.x
    y = twi.linear.y
    br.sendTransform((x,y,0.0),(0.0,0.0,0.0,1.0),rospy.Time.now(),"robot0/goal","robot0/world")

if __name__ == '__main__':
    try:
        rospy.init_node('tf_boadcaster0',anonymous=True)
        rospy.Subscriber("/robot_0/base_pose_ground_truth",Odometry,worldframe)
        rospy.Subscriber("/robot_0/goal",Twist,goalframe)
        rospy.spin()
    except rospy.ROSInterruptException: pass
```

testclient.py

```python
#! /usr/bin/env python

import rospy
import tf
from geometry_msgs.msg import Twist
from geometry_msgs.msg import Vector3
from std_msgs.msg import Float64
import roslib
import actionlib

import as5831_rosbot.msg

def goal0_client():
    client=actionlib.SimpleActionClient('robot0_check_goals',as5831_rosbot.msg.goalStatusAction)
    client.wait_for_server()
    goal = as5831_rosbot.msg.goalStatusGoal(x=3,y=12)
    client.send_goal(goal)
    client.wait_for_result()
    return client.get_result()

if __name__=='__main__':
    try:
        rospy.init_node('test_client')
        result = goal0_client()
        print result.thereOrNot
    except rospy.ROSInterruptException:
        print "program interrupted before completion"
```

Goal0.py

```python
#!/usr/bin/env python
import rospy
import tf
from geometry_msgs.msg import Twist
from geometry_msgs.msg import Vector3
from std_msgs.msg import Float64
import roslib
import actionlib
import as5831_rosbot.msg

currentGoal = Vector3(3.0,12.0,0.0)

def goal0_client():
    global currentGoal

    client=actionlib.SimpleActionClient('check_goals',as5831_rosbot.msg.goalStatusAction)
    client.wait_for_server()
    goal = as5831_rosbot.msg.goalStatusGoal(x=currentGoal.x,y=currentGoal.y)
    client.send_goal(goal)
    client.wait_for_result()
    return client.get_result()

# creates a fixed goal frame in tf
def goalPub():
    global currentGoal
    goalCount = 0
    goalinworld = rospy.Publisher('robot_0/goal',Twist,queue_size=10)
    goal = Twist()
    goal.linear = Vector3(currentGoal.x,currentGoal.y,0.0)
    goal.angular = Vector3(0.0,0.0,0.0)

    rate = rospy.Rate(10.0)
    while not rospy.is_shutdown():
        result = goal0_client()
        if result.robot0_thereOrNot == 1:
            goalCount = goalCount + 1

        if goalCount>=1:
            currentGoal.x = 0.0
            currentGoal.y = 0.0
        else:
            currentGoal.x = 3.0
            currentGoal.y = 12.0

        goal.linear = Vector3(currentGoal.x,currentGoal.y,0.0)
        goal.angular = Vector3(0.0,0.0,0.0)

        rospy.loginfo(goalCount)
        rospy.loginfo(goal.linear)
        goalinworld.publish(goal)
        rate.sleep()

if __name__ == '__main__':
    try:
        rospy.init_node("robot0_goal")
        goalPub()
    except rospy.ROSInterruptException: pass
```

Tangentbug0.py

```python
#!/usr/bin/env python

import rospy
import tf
import math
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Vector3
from std_msgs.msg import UInt32

maxRange = 5.0

goalVisible = 0

discThresh = 5.0

scan = LaserScan()

def scanUpdate(lScan):
    global scan
    scan = lScan

def tangentbug():
    global goalVisible
    global scan
    goalfromlaser = rospy.Publisher('robot_0/robot2goal',Twist,queue_size=10)
    robotController = rospy.Publisher('robot_0/cmd_vel',Twist,queue_size=10)

    listener = tf.TransformListener()
    rate = rospy.Rate(1.0)
    while not rospy.is_shutdown():
        try:
            (trans,rot) = listener.lookupTransform('robot0/trueOdom','robot0/goal',rospy.Time(0))
            twi = Twist()
            twi.linear.x = trans[0]
            twi.linear.y = trans[1]
            twi.linear.z = trans[2]
            twi.angular = Vector3(0.0,0.0,0.0)

            control = Twist()
            control.linear = Vector3(0.0,0.0,0.0)
            control.angular = Vector3(0.0,0.0,0.0)
            angle2goal = math.atan2(trans[1],trans[0])

            sensorIndex = int((angle2goal-scan.angle_min)/scan.angle_increment)

            # check if we can see the goal directly. Move towards it if we can
            if (scan.ranges[sensorIndex] >= maxRange):
                goalVisible = 1
                if (angle2goal > 0):
                    control.linear.x = 0.3
                    control.angular.z = 0.2
                elif (angle2goal < 0):
                    control.linear.x = 0.3
                    control.angular.z = -0.2
                else:
                    control.linear.x = 0.3
                    control.angular.z = 0.0
            else:
                goalVisible = 0
```

```
63
64          if goalVisible == 0:
65              bestAngle = 0.0
66              besti = 0
67              bestDist = 10000.0
68              realAngle = 0.0
69
70              for i in range(len(scan.ranges)):
71                  # check for discontinuties within a specified threshold
72                  if (i>0) and (abs(scan.ranges[i]-scan.ranges[i-1]) > discThresh):
73                      # output the index for the discontinuity and the angle value and the distance to that discontinuity
74                      discDist = scan.ranges[i]
75                      if discDist==float('Inf'):
76                          discDist = scan.range_max
77                      dAng = scan.angle_min + i * scan.angle_increment
78                      xDist = discDist * math.sin(dAng)
79                      yDist = discDist * math.cos(dAng)
80                      heurDist = math.sqrt((twi.linear.x-xDist) ** 2 + (twi.linear.y-yDist) ** 2)
81                      if ((heurDist + discDist) < bestDist):
82                          bestDist = heurDist + discDist
83                          bestAngle = dAng
84                          besti = i
85
86
87              if ((bestAngle) > 0):
88                  control.linear.x = 0.2
89                  control.angular.z = 0.5
90              elif ((bestAngle) < 0):
91                  control.linear.x = 0.2
92                  control.angular.z = -0.5
93              else:
94                  control.linear.x = 0.2
95                  control.angular.z = 0.0
96
97
98              if (besti > 90) and (besti < (len(scan.ranges)-90)):
99                  if scan.ranges[besti+20] < 2.0:
100                     control.linear.x = 0.2
101                     control.angular.z = 0.5
102                 elif scan.ranges[besti-20] < 2.0:
103                     control.linear.x = 0.2
104                     control.angular.z = -0.5
105             elif (besti > 90) and (besti < (len(scan.ranges)-90)):
106                 if scan.ranges[besti+30] < 2.0:
107                     control.linear.x = 0.2
108                     control.angular.z = 0.5
109                 elif scan.ranges[besti-30] < 2.0:
110                     control.linear.x = 0.2
111                     control.angular.z = -0.5
112
113
114         j = int(len(scan.ranges)/2) - 70
115         m = int(len(scan.ranges)/2) - 10
116         k = int(len(scan.ranges)/2) + 70
117         n = int(len(scan.ranges)/2) + 10
118
119         for i in range(j,m):
120             if (scan.ranges[i] < 2.5):
121                 control.linear.x = 0.0
122                 control.angular.z = 0.5
123         for i in range(n,k):
124             if (scan.ranges[i] < 2.5):
125                 control.linear.x = 0.0
126                 control.angular.z = -0.5
```

```
127
128
129                 if math.sqrt((twi.linear.x) ** 2 + (twi.linear.y) ** 2) < 0.5:
130                     control.linear.x = 0.0
131                     control.linear.y = 0.0
132
133                 robotController.publish(control)
134                 goalfromlaser.publish(twi)
135             except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):
136                 continue
137
138         rate.sleep()
139
140 if __name__ == '__main__':
141     try:
142         rospy.sleep(2.0)
143         rospy.init_node('tf_robot0_as5831_rosbot')
144         rospy.Subscriber("robot_0/base_scan",LaserScan,scanUpdate)
145         tangentbug()
146         rospy.spin()
147     except rospy.ROSInterruptException: pass
```