COMP3431

Robotic Software Architecture

Assignment 2: Report

Nathan ADLER z3327870
Aneita YANG z5017723

November 9, 2015

Contents

1 Introduction

In this assignment, both the hardware and software aspects of robotics are explored. The overall objective was to create a robot that could navigate autonomously in an outdoor environment, whilst avoiding any obstacles. An existing ground vehicle platform developed by student society CREATE UNSW was used as a base for testing.

To achieve the objective, the robot is equipped with a GPS and compass (using an Android phone with ROS). A laser scanner is also attached to the front of the robot, gathering information about the robot's immediate surroundings.

1.1 Modules

1.1.1 Hardware

The key hardware elements that combine to provide sensory information, networking and control are as follows:

- A laptop, to run the ROS core and all nodes that interpret sensory information and output control commands
- A LIDAR that provides a 2D field of view of any nearby obstacles
- A phone that provides GPS and compass data
- A motor controller system and motors that drive the robot
- A real-time kinematic (RTK) GPS positioning module that provides centimetre-level accuracy in good conditions

1.1.2 Software

On the software side, six nodes run in conjunction to operate the robot:

`dest_sender` keeps track of the robot's remaining waypoints.

`gps_drive` calculates and publishes the direction in which the robot needs to travel to reach its destination.

`rtk_gps_pub` publishes the an optional precise position based on real time kinematic GPS solutions with centimetre-level accuracy.

`laser_safe` publishes movement messages, either directly to the bot's destination, or to avoid an obstacle.

`motordata_arduino_send` converts movement messages into serial messages to send to the Arduino that controls the motord driver and drives the robot.

`sick_tim` reads data from the SICK TiM551 2D laser scanner and publishes the data as a `laser_scan` ROS message.

2 Hardware

2.1 Mechanical Structure

The Unmanned Ground Vehicle (UGV) was developed to serve as an autonomous platform for the research and testing of robotic systems. It measures 1 m long, 0.5 m wide, and when fully loaded weighs approximately 50 kg. The robot was constructed using parts from an electric wheelchair and a custom-design welded steel chassis, as shown in Figure 1.

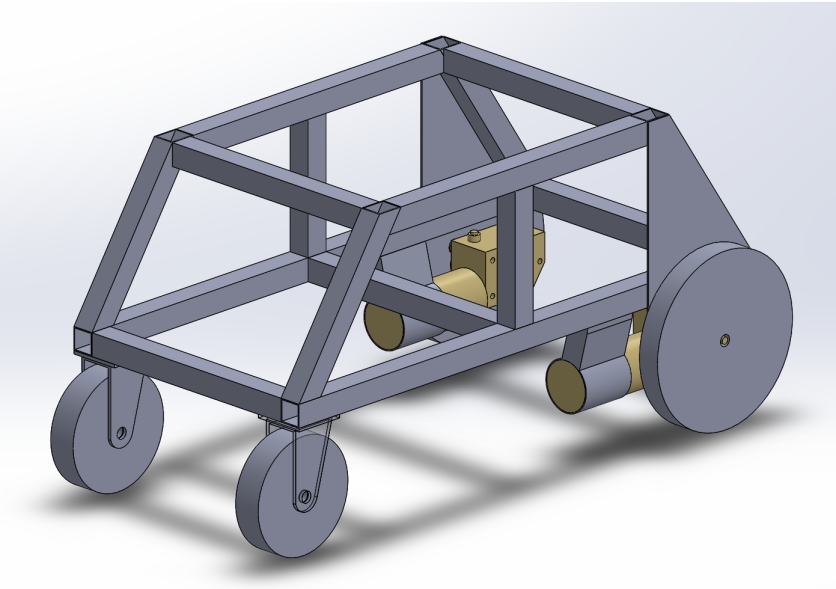

Figure 1: **CAD model of chassis of UGV**

2.2 Specifications and Drivers

2.2.1 SICK TiM551 2D Laser Scanner

The 2D laser scanner (shown in Figure 2) is rated outdoor IP67, allowing for use in a variety of weather conditions including bright sunlight. The sensor provides readings up to a maximum range of 10 m, giving enough perspective to traverse routes in terrain with complex and even dynamic obstacles. The following specifications detail the capabilities of the sensor:

- Aperture angle: 270 °
- Scanning frequency: 15 Hz
- Angular resolution: 1 °
- Operating range: 0.05-10 m
- Operating voltage: 9-28 V DC

Figure 2: **SICK TiM551 2D Laser Scanner**

- Power consumption: 3 W

The device is mounted in a static location on the front of the UGV in the horizontal plane approximately 300 mm above ground level. In this position, the view is partially obscured to the sides and behind the laser scanner by the UGV chassis, and therefore the view is limited to a 150 ° sweep to the front of the robot in software.

To obtain data from the laser scanner, the open source `sick_tim` ROS package is used. The package provides a launch file `sick_tim551_2050001.launch`, which reads range data into a `sensor_msgs/LaserScan.msg` that is published every scan frame.

2.2.2 Android phone GPS and compass

The robot is equipped with an Android phone running the `android_sensors_driver` application called ROS All Sensors. The app publishes:

- Camera images: `sensor_msgs/CompressedImage` and `sensor_msgs/Image`
- Fluid pressure data: `sensor_msgs/FluidPressure`
- Illuminance data: `sensor_msgs/Illuminance`
- Accelerometer: `sensor_msgs/Imu`
- Magnetic field: `sensor_msgs/MagneticField`
- GPS fixes: `sensor_msgs/NavSatFix`
- Temperature data: `sensor_msgs/Temperature`

During operation, camera streaming is not enabled to ensure there is enough bandwidth and processing available for proper operation of all software modules. The GPS receiver is able to use American GPS and Russian GLONASS satellites, and provides a ”fix” solution at a rate

of 1 Hz. The fix solution is formatted as a `sensor_msgs/NavSatFix.msg` (see Appendix A) All other sensor data is published at a rate of 20 Hz.

2.2.3 RTK GPS Module

A specialised GPS device (shown in Figure 3) is mounted on the UGV for precise positioning. It consists of an antenna with inbuilt raw receiver, and an Odroid U3 with a 3G dongle. The open source software computes centimetre-accurate positions at 2 Hz and is configured to output solutions over TCP connection through the VPN. The latitude and longitude solutions are then received and published as a `NavSatFix.msg` by `rtk_gps_pub`.

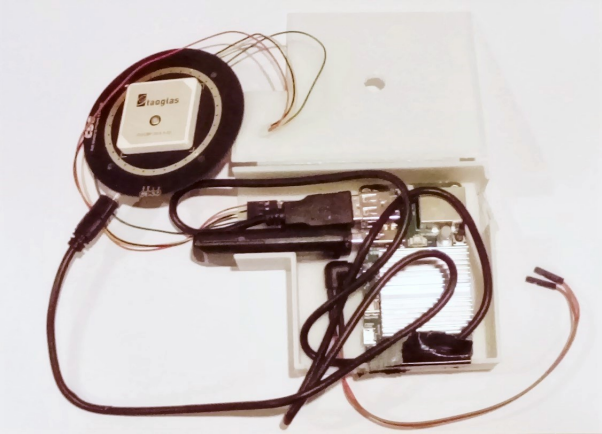

Figure 3: **RTK GPS precise positioning device mounted on UGV**

2.2.4 Arduino and Motor Driver

The dual-channel motor driver converts a pulsed signal to pulse-width modulated (PWM) output to drive the motors in either direction. The pulse range from control ranges from 1000-2000 µs, where 1500 is off, 2000 is full forward and 1000 is full reverse. The pulse signals are generated by the Arduino Mega 2560 microcontroller, which is connected via USB to the laptop. The microcontroller is programmed to receive messages in a custom format as follows:

[SAADBB:CC]

where S is direction of bridge A, either '+' or '-', AA is the PWM of bridge A, from 00 to FF, D is direction of bridge B, either '+' or '-', BB is the PWM of bridge B, also from 00 to FF, and CC is the checksum of the command. The Arduino then scales and converts the command into the appropriate pulse signal timing to output to the dual-channel motor driver.

The Arduino program also includes motor controls that smooth changes in speed and direction to avoid damaging the motors, and a watchdog timer that stops the motors if no new command has been received for 500 ms.

2.3 Integration

The following hardware is contained on board:

- Dell Latitude E6400 notebook
- SICK TiM551 2D laser scanner
- DLink DSL 2750B N300 Modem Router
- Sony Xperia Z3 compact
- Arduino Mega 2560 R3
- Sabertooth 2X25 regenerative motor driver
- 2 x 12V lead acid battery
- 2 x electric wheelchair motor

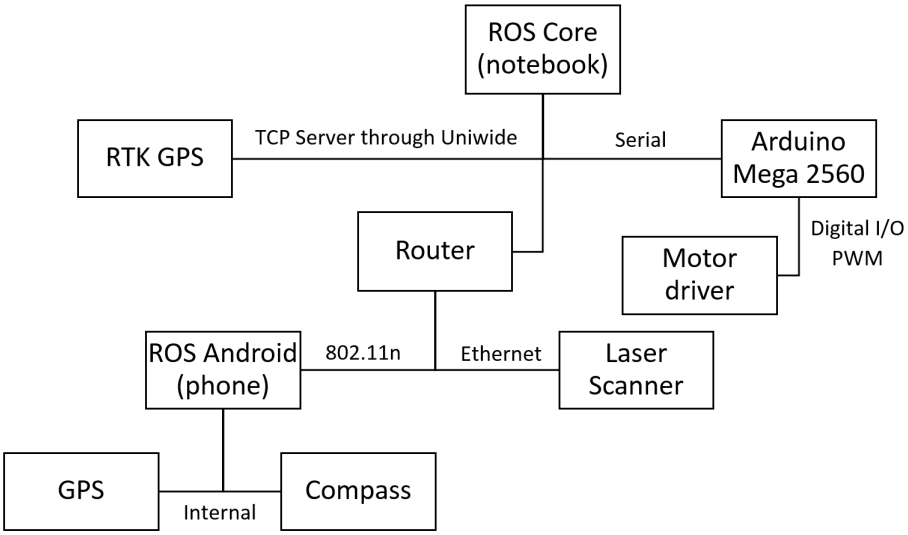

Figure 4: **Hardware map of UGV**

The Dell notebook is installed with Ubuntu 14.04 (a.k.a. "Trusty Tahr") and ROS Indigo. Sensors from the phone and laser scanner are integrated in a Local Area Network through a combination of Ethernet and wireless connections. A VPN is installed for remote communication over the internet between networked devices, including a precise positioning GPS device developed for an undergraduate thesis. 24 V power from the two lead acid batteries is supplied to the motor driver to power the DC geared motors and integrated solenoid brakes. 12 V from one of the batteries is segmented to provide a separate power source for the laser scanner and the router.

Note: this hardware configuration was not set up prior to undertaking this assignment (excluding the chassis, motors and motor driver); all hardware development and integration was conducted as part of the project.

3 Software

Three main modules run in conjunction to govern the UGV: planner, localisation and waypoint traversal.

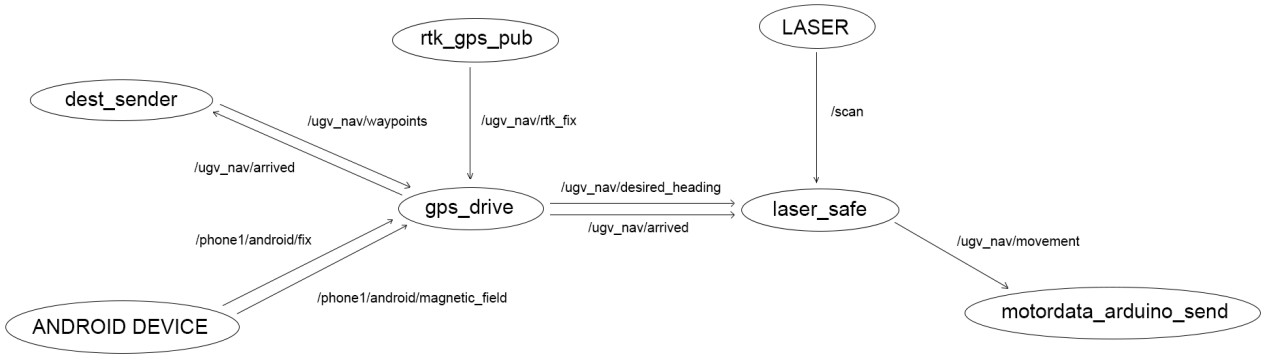

Figure 5: **Rqt graph.**

3.1 Planner

The planner module is responsible for keeping track of the robot's waypoints and informs the robot of its next destination.

The `dest_sender` node parses a file containing the GPS coordinates of waypoints and turns each latitude-longitude pair into a NavSatFix message. Each waypoint is stored in a list (as a NavSatFix message), with the head of the list being the robot's next destination. `dest_sender` publishes this message to the `/ugv_nav/waypoints` topic for other modules to subscribe to.

`dest_sender` subscribes to the `/ugv_nav/arrived` topic, which signals when the robot has reached its destination. Messages which are published to the `/ugv_nav/arrived` topic trigger a callback which removes the current waypoint from the list (i.e. the head of the list). If the robot has not reached its final destination and the list is not empty, the robot's next destination is published.

3.2 Localisation

3.2.1 UGV position and heading determination

The localisation module is responsible for determining the direction in which the robot needs to travel to reach its goal. The robot. In order to determine this direction, the robot must be aware of its position in GPS coordinates, its bearing from true north and the GPS coordinates of its destination.

For this assignment, the `gps_drive` node performs these required calculations. In particular, the `gps_drive` node subscribes to the Android device's GPS fixes and magnetic fields (published on the `/phone1/android/fix` and `/phone1/android/magnetic_field` topics respectively). The robot also has the option to receive real-time kinematic GPS data through `/ugv_nav/rtk_fix`, providing a much more accurate position from an external GPS module in good conditions. A filter is implemented that determines whether the RTK GPS data is suitable for use by ensuring that the five latest readings (2.5 seconds of data) are within an acceptable area. If the RTK GPS does not have a good fix solution, the positioning calculations revert to using the phone GPS.

The GPS coordinates of the robot's destination are published to the `/ugv_nav/waypoints` topic by the `dest_sender` node, which `gps_drive` also subscribes to.

3.2.2 Waypoint bearing calculation

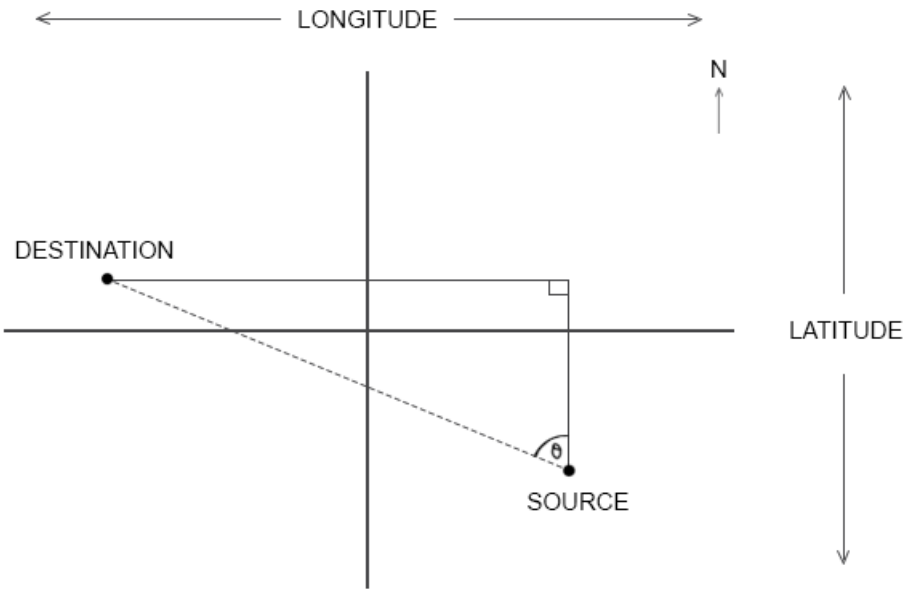

Figure 6: **Calculation of the bearing of the destination from the robot, if the robot is facing true north.**

Calculating the bearing of the destination from the robot is dependent on the GPS coordinates of both the robot and the destination.

$$\begin{aligned} \Delta latitude &= \text{destination\_latitude - robot\_latitude} \\ \Delta longitude &= \text{destination\_longitude - robot\_longitude} \end{aligned} \tag{1}$$

Supposing the robot is facing True North, and if β is the bearing of the destination from the robot then:

$$\beta = \tan^{-1}\left(\frac{\Delta longitude}{\Delta latitude}\right) \tag{2}$$

If θ is the angle the robot must turn to reach its destination, as in Figure 6, then $\theta = \beta$.

In the cases where the robot is not facing true north, then:

$$\theta = (-\alpha) + \beta \tag{3}$$

where:

$$\begin{aligned} \beta &= \tan^{-1}(\frac{\Delta longitude}{\Delta latitude}) \\ \alpha &= \tan^{-1}(\frac{x}{y}) \end{aligned} \tag{4}$$

where x and y are magnetic field strength to the top and right edges of the mobile phone respectively.

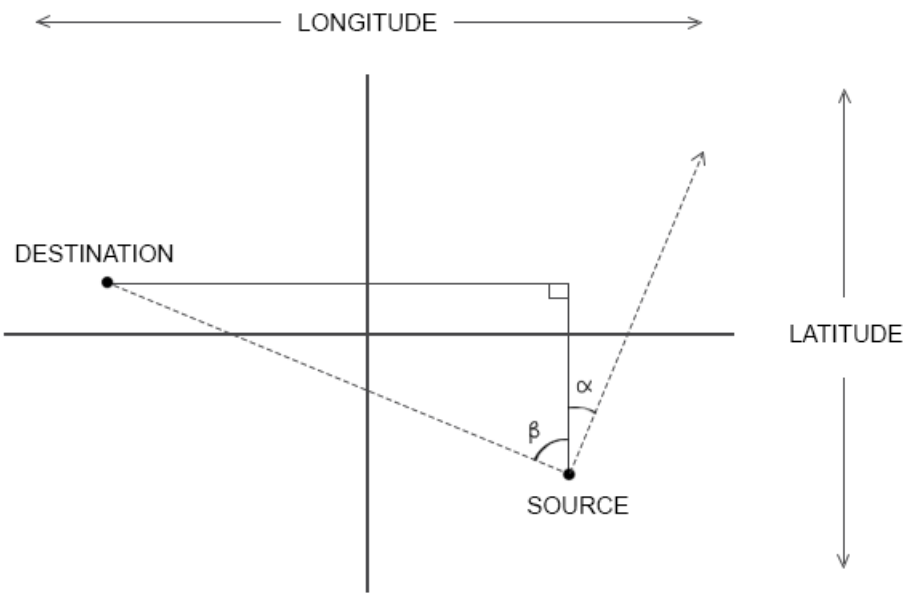

Figure 7: **Calculation of the bearing of the destination from the robot where the robot is not facing true north. NOTE: In this example, α is positive and β is negative.**

Since at the latitude of Sydney, one degree of latitude is approximately 111 km:

$$\text{distance\_to\_destination} = 111000\sqrt{\text{difference\_lat}^2 + \text{difference\_long}^2} \tag{5}$$

If the distance to the destination is less than 5 (i.e. the robot is within 5 metres of its destination), it is considered to have arrived at its destination. At this stage, the `gps_drive` node will publish a message to the `/ugv_nav/arrived` topic, signalling to the `dest_sender` that the robot has reached its destination and that a new destination is required.

Because of the inaccuracies of the compass and the GPS, the vehicle may have a tendency to deviate from the straight line path between waypoints. This can result in the robot wandering off the most suitable path. To counter this, a bias toward the straight line path is added to direct the robot back on track. The bias is made proportional to the perpendicular distance from the straight line path according to the following equation:

$$bias = k * \text{distance\_to\_destination} * sin(\text{desired\_heading} - \text{start\_to\_dest\_heading}) \tag{6}$$

where k is the proportional constant, chosen to be 2, and `start_to_dest_heading` is the heading from either the starting position or the last waypoint to the current destination. For example, if the position is calculated to have deviated 5 m from the straight line path, then the robot will adjust its bearing 10 ° in addition to the new straight line bearing.

Following these calculations, the `gps_drive` node publishes the most applicable bearing (as a std_msgs::Float32 message) for the robot to travel. The waypoint traversal module then performs the applicable actions based on this bearing.

3.3 Waypoint Traversal

The waypoint traversal module is responsible for physically getting the robot to its destination. The `gps_drive`, `laser_safe` and `motordata_arduino_send` nodes work in conjunction to achieve this goal.

3.3.1 Obstacle Avoidance

To detect any obstacles in the robot's path, a laser scanner is attached to the front of the robot.

Given a bearing to travel along, the `laser_safe` node will take one of two courses of action. If there is no obstacle in front of the robot, the robot will drive in the direction it was instructed to by the `gps_drive` node. If an obstacle is present, however, the robot's first priority is to avoid colliding with the obstacle.

Note that for the purposes of this assignment, the robot is instructed to scan for obstacles within a 150 ° field-of-view. That is, obstacles further than 75 ° to the left or right of the centre of the robot are deemed as harmless and are ignored.

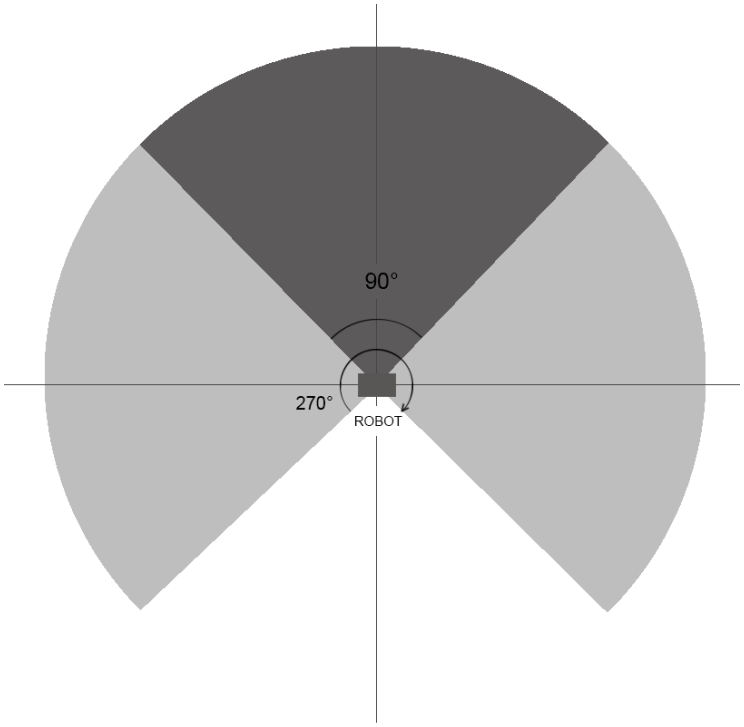

Figure 8: **Range of laser scanner for Iterations 1 and 2.**

First Iteration

The first iteration of obstacle avoidance involves bringing the robot to a complete stop when an obstacle enters the view of the laser scanner. As the UGV's main task is to drive autonomously in outdoor environments, this method proves useful for moving obstacles (e.g. people) that may enter and exit the frame quickly.

Second Iteration

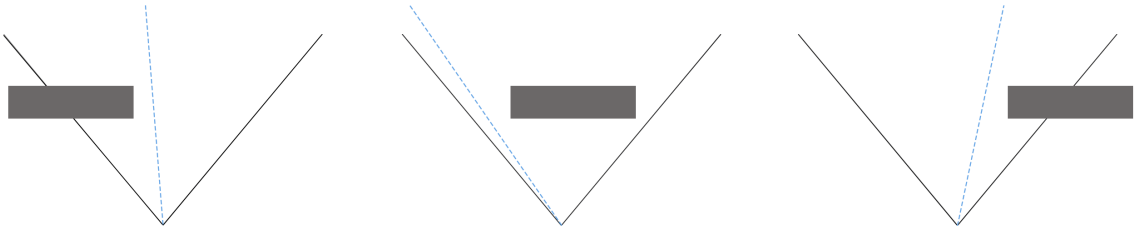

Figure 9: **Obstacle avoidance paths for Iteration 2.**

The second iteration of obstacle avoidance attempts to drive around obstacles as they enter the danger zone (i.e. 45°left and right of the centre of the robot, within 2 m). The robot does this by picking the closest 'safe' spot to the left of the first obstacle, if possible, and drives towards it. In cases where the obstacle cuts out of the left-hand side of the laser's frame, the closest 'safe' spot to the right is chosen.

Once the robot is clear of all obstacles, it resumes driving towards its destination.

Final Iteration

To handle the case where multiple distinct obstacles are visible, and various distances away need to be taken into account with the width of the robot to provide a safe bearing, the previous obstacle avoidance method is improved by finding the best available heading that is passable. The algorithm undergoes the following steps:

1. Populate an array the length of the number of ranges in the scan range with '1' for visible obstacles within 3 m and '0' for clear space
2. Widen the bearing occupied by any visible obstacles within the array according to their distance away from the robot, so that a bearing buffer equivalent of 0.7 m on either side of the detected obstacle is occupied, using the formula in (7).
3. Choose the available bearing that is closest to the desired bearing calculated in (3).
    - If no bearing is available, do not publish any command for movement.

This process is shown in Figure 10. Code for the algorithm can be found in Appendix B.

Additionally, if any obstacle comes within 0.7 m of the robot, a safety stop is enacted and the robot will not move.

$$\begin{aligned} \text{angle\_buffer} &= sin^{-1}(\frac{\text{SAFE\_DISTANCE}}{range}) \\ \text{where SAFE\_DISTANCE} &= 0.7 \end{aligned} \tag{7}$$

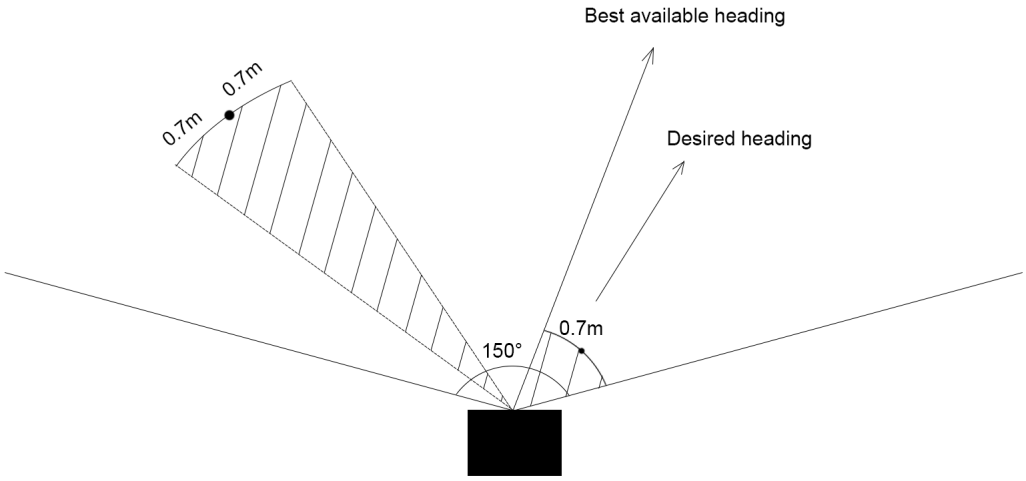

Figure 10: **Obstacle avoidance for Final Iteration.**

3.3.2 Motor Control

To convert a motion heading into motor commands, a motor behaviour transform is enacted to mimic joystick control for a differentially steered vehicle, such as an electric wheelchair. The function responsible for carrying out this transformation is recorded in Appendix C. The variation of motor power with respect to heading is mapped in Figure 11.

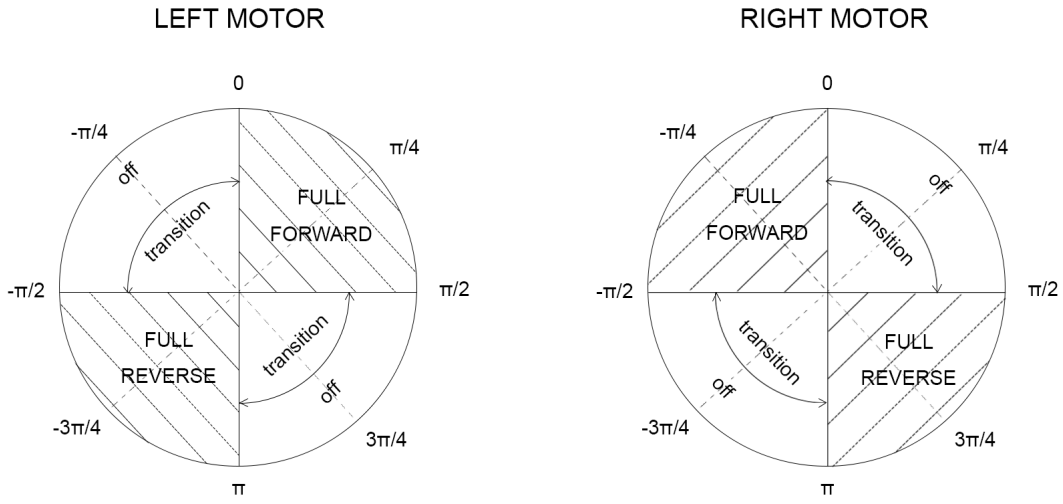

Figure 11: **Motor controls.**

To direct the robot in the direction of the next waypoint according to the calculated heading dependent on bias corrections and obstacle avoidance, a proportional controller is implemented for steering. The local desired heading θ is halved and then constrained to $\pm 30\,^\circ$ to ensure a controlled turn regardless of the difference between the current and desired heading. This guarantees the robot does not drive backwards or carry out a sharp pivot which would risk the robot crashing into obstacles it cannot see. At the same time, the proportional controller allows for subtle changes in direction through varying power to the motors that differentially steer the robot. As the current location and desired heading of the robot are updated at regular intervals, the steering heading of the robot will gradually adjust accordingly.

For testing, the magnitude of the speed control was fixed at 70%. When traversing between waypoints, if a valid heading was available (no obstacles within 0.7 m, no impassable barriers) then a motion command is published. This motion command is updated at 10 Hz. The command was formatted according to the requirements of the Arduino motor driver bridge (see 2.2.4) and sent over serial.

4 Results

For testing, the `gps_config` file was loaded with a series of 10 waypoint coordinates starting on the lower campus main walkway of UNSW outside the Tyree Energy Technologies building, finishing at the bottom of the Scientia building. The robot then was allowed to drive autonomously as it attempted to complete the journey between the two points. The robot successfully navigated to its destination and then returned, although required some assistance in certain situations.

The demonstration for the assignment performed well in large open areas with no tall surrounding objects. The main causes for incorrect behaviour were poor GPS fixes, blind spots in the laser scanner and interference with the laser scanner.

Inaccurate GPS Fixes

Waypoint traversal relies heavily on the robot being able to pinpoint its precise position. In sheltered areas, or areas with tall buildings or trees, GPS fixes can be inaccurate. Such inaccuracies impact on the accuracy of the calculations performed. The bearing of the destination from the robot's position can be incorrect. The distance the robot must drive before reaching its destination can also be wrong. When working with unmanned vehicles, it is particularly important that these calculations are precise and accurate. In circumstances where the robot had a precise RTK GPS fix solution, mainly in areas not in proximity to tall buildings on multiple sides, the driving of the robot was significantly improved.

Laser Scanner Blind Spots

The laser scanner is positioned on the front of the robot, on its base. Sitting at a height of 30 cm above the ground, the laser scanner scans on a level plane. Obstacles shorter than 30 cm, that lie outside of the laser's view, are essentially driven over or driven through. During the demonstration, the robot drove into a brick wall which was just below the laser scanner's view. The robot was also sometimes confused when variations in the gradient of the ground saw it to believe that obstacles were nearby, such as on grassed areas where the laser scanner would detect the grass. The laser scanner's limited view also meant that care had to be taken when the robot was undergoing sharp turns, and this limited its range of motion.

Laser Scanner Interference

Obstacle avoidance relies entirely on data received from the laser scanner. In areas with reflective surfaces (e.g. mirrors and glass), the reflection interferes with the laser, causing the robot to think there are obstacles even when the path is clear. This caused the robot to think that certain paths were blocked and try to go around non-existent obstacles. A fix was attempted that involved filtering out detected obstacles the size of only one point, which somewhat improved navigation.

5 Future Work and Improvements

A number of improvements are possible in future revisions of the software on the UGV to enhance navigation capabilities and increase its potential as a platform for research and development. These include:

- Adding a depth sensor on front such as Kinect to sense ledges and different terrain.
- Adding odometry to better track the movement of the robot.
- Implementing an EKF with the odometry to improve errors in positioning.
- Adding a sensor on the rear to allow for autonomous reversing.
- Implementing map-based localisation and navigation for better path planning.
- Extending navigation into indoor environments and manage transition between outdoor and indoor settings.

Appendix

A sensor_msgs/NavSatFix.msg

```
Header header

/* Satellite fix status information */
NavSatStatus status

/* Latitude [degrees]. Positive is north of equator; negative is south. */
float64 latitude

/* Longitude [degrees]. Positive is east of prime meridian; negative is west. */
float64 longitude

/*
Altitude [m]. Positive is above the WGS 84 ellipsoid
(quiet NaN if no altitude is available).
*/
float64 altitude

/*
Position covariance [m^2] defined relative to a tangential plane
through the reported position. The components are East, North, and
Up (ENU), in row-major order.
Beware: this coordinate system exhibits singularities at the poles.
*/
float64[9] position_covariance

/*
If the covariance of the fix is known, fill it in completely. If the
GPS receiver provides the variance of each measurement, put them
along the diagonal. If only Dilution of Precision is available,
estimate an approximate covariance from that.
*/
uint8 COVARIANCE_TYPE_UNKNOWN = 0
uint8 COVARIANCE_TYPE_APPROXIMATED = 1
uint8 COVARIANCE_TYPE_DIAGONAL_KNOWN = 2
uint8 COVARIANCE_TYPE_KNOWN = 3

uint8 position_covariance_type
```

B Obstacle Avoidance Function

```
1  int Laser::getBestHeading(vector<float> ranges, int desired_heading_transform)
       const {
2     // note: no value in ranges can be less than SAFE_DISTANCE for this function
          to work
3     int obstacles[RANGES] = {0};
4     int best_heading = -1; // no heading currently available
5     double angle_buffer_rad;
6     int angle_buffer_deg = 0;
7     // populate obstacles array by padding each obstacle reading on both sides
8     // first, sweep one way, right to left
9     for (int i = MIN_RANGE; i <= MAX_RANGE; ++i) {
10       if (ranges[i] != 0.0 && ranges[i] < DETECT_RANGE) {
11          angle_buffer_rad = asin(SAFE_DISTANCE/ranges[i]);
12          angle_buffer_deg = fmax(angle_buffer_deg,floor(angle_buffer_rad *
               180/M_PI));
13       }
14       if(angle_buffer_deg>0){
15          obstacles[i] = 1;
16          angle_buffer_deg--;
17       }
18    }
19    // then, sweep the other way
20    angle_buffer_deg = 0;
21    for (int i = MAX_RANGE; i>=MIN_RANGE; --i) {
22       if (ranges[i] != 0.0 && ranges[i] < DETECT_RANGE) {
23          angle_buffer_rad = asin(SAFE_DISTANCE/ranges[i]);
24          angle_buffer_deg = fmax(angle_buffer_deg,floor(angle_buffer_rad *
               180/M_PI));
25       }
26       if(angle_buffer_deg>0){
27          obstacles[i] = 1;
28          angle_buffer_deg--;
29       }
30    }
31    //then, get the viewable angle free of obstacles closest to desired heading
32    for (int i = MIN_RANGE,abs_diff = 1000; i <= MAX_RANGE; ++i) {
33       if (obstacles[i]==0){
34          if(fabs(i-desired_heading_transform)<abs_diff){
35             best_heading = i;
36             abs_diff = fabs(i-desired_heading_transform);
37          }
38       }
39    }
40    return best_heading;
41 }
```

C Movement Message to Motor Control Function

```
1  void Motornav_Com::sendMovement() {
2
3     if(r_d<0.1)
4        return;
5     // new driving maths coming from old UGV_sketch
6     if(theta_d<=M_PI_4&&theta_d>=-3*M_PI_4){
7        rightMotorSign = '+';
8     } else {
9        rightMotorSign = '-';
10    }
11    if(theta_d<=3*M_PI_4&&theta_d>=-M_PI_4){
12       leftMotorSign = '+';
13    } else {
14       leftMotorSign = '-';
15    }
16    leftMotorVal = fmin(M_PI_4,fmin(fabs(theta_d+M_PI_4),
          fabs(theta_d-3*M_PI_4)))*255/M_PI_4;
17    rightMotorVal = fmin(M_PI_4,fmin(fabs(theta_d+3*M_PI_4),
          fabs(theta_d-M_PI_4)))*255/M_PI_4;
18
19    leftMotorVal = fmin(leftMotorVal*r_d,255);
20    rightMotorVal = fmin(rightMotorVal*r_d,255);
21
22    char buffer[MOTORDRIVER_COMBUFFER_LENGTH+1];
23    char message[MOTORDRIVER_COMBUFFER_MSGLENGTH+1];
24    snprintf(message, MOTORDRIVER_COMBUFFER_MSGLENGTH+1, "%c%2.2X%c%2.2X",
          leftMotorSign, leftMotorVal, rightMotorSign, rightMotorVal);
25    unsigned char checksum = calculateChecksum(message,
          MOTORDRIVER_COMBUFFER_MSGLENGTH);
26    snprintf(buffer,MOTORDRIVER_COMBUFFER_LENGTH+1,"[%s:%2.2X]",message,checksum);
27    printf("%.*s\n", MOTORDRIVER_COMBUFFER_LENGTH, buffer);
28
29    // Write these bytes to the Com Port
30
31    //ROS_INFO_STREAM("theta " << theta_d << " mag " << r_d);
32    comPort.write(buffer, MOTORDRIVER_COMBUFFER_LENGTH);
33    comPort.flush();
34 }
```