COMP3431

Robotic Software Architecture

Assignment 2: Report

Nathan ADLER
Aneita YANG

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 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, and weighs approximately 50 kg. The robot was constructed using parts from an electric wheelchair and a custom-design welded steel chassis. 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

The Dell notebook was installed with Ubuntu 14.04 (a.k.a. "Trusty Tahr") and ROS Indigo. Sensors from the phone and laser scanner were integrated in a Local Area Network through a combination of Ethernet and wireless connections. A VPN was installed for remote communication over the internet between networked devices, including a precise positioning GPS device developed for an undergraduate thesis.

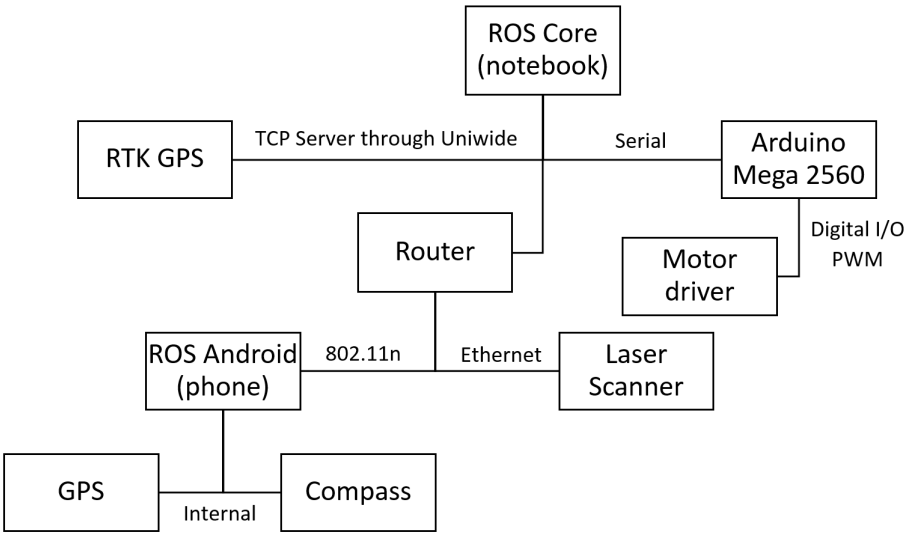

Figure 1: Hardware map of UGV

1.1.2 Software

On the software side, five/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.

1.2 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.

1.3 Localisation

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.

To obtain this information, the robot is equipped with an Android phone running the `android_sensors_driver` app which 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`

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 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.

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

$$\Delta latitude = \text{destination\_latitude} - \text{robot\_latitude}$$
$$\Delta longitude = \text{destination\_longitude} - \text{robot\_longitude}$$

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

$$\beta = \tan^{-1}(\frac{\Delta longitude}{\Delta latitude}),$$

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

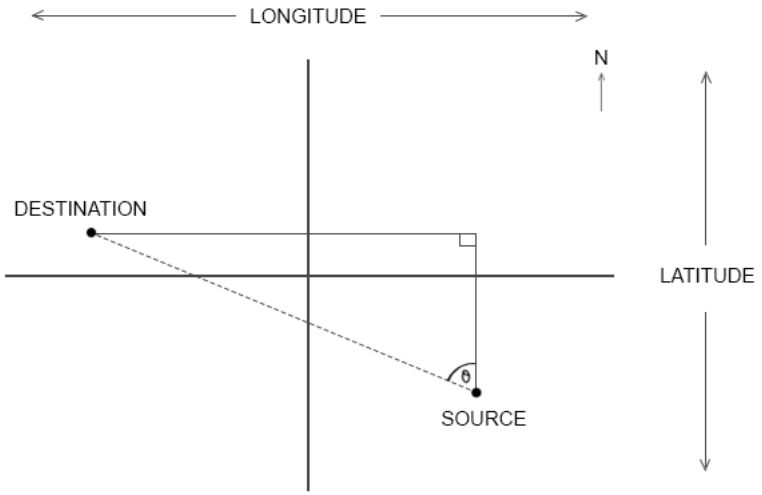

Figure 2: **Calculation of the bearing of the destination from the robot, if the robot is facing true north.**

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

$$\theta = (-\alpha) + \beta$$

where:

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

$$\alpha = \tan^{-1}(\frac{\Delta x}{\Delta y})$$

and Δx and Δy are given from the Android device's magnetic field readings.

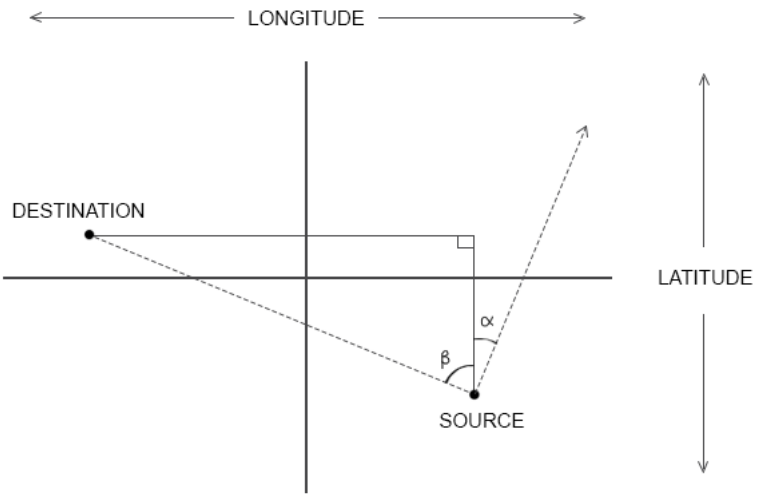

Figure 3: **Calculation of the bearing of the destination from the robot where the robot is not facing true north. NOTE: In this example, α is positive and β is negative.**

As the ???th decimal place provides up to ??? precision:

$$\text{distance\_to\_destination} = 111,000 * \sqrt{(\text{difference\_lat})^2 + (\text{difference\_long})^2}$$

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.

As the robot obtains GPS fixes frequently, it is sensitive to small changes to its position. The frequent updating of the robot's GPS fix results in abrupt movement changes as the robot attempts to face the bearing to its destination. In order to reduce the amount of sudden

directional changes, rather than continually looking for the most direct path to its goal, the direction the robot decides to travel is biased towards a 'smooth path'.

TODO: Explain straight line bias calculations

Once these biases have been taken into consideration, 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.

1.4 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.

1.4.1 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.

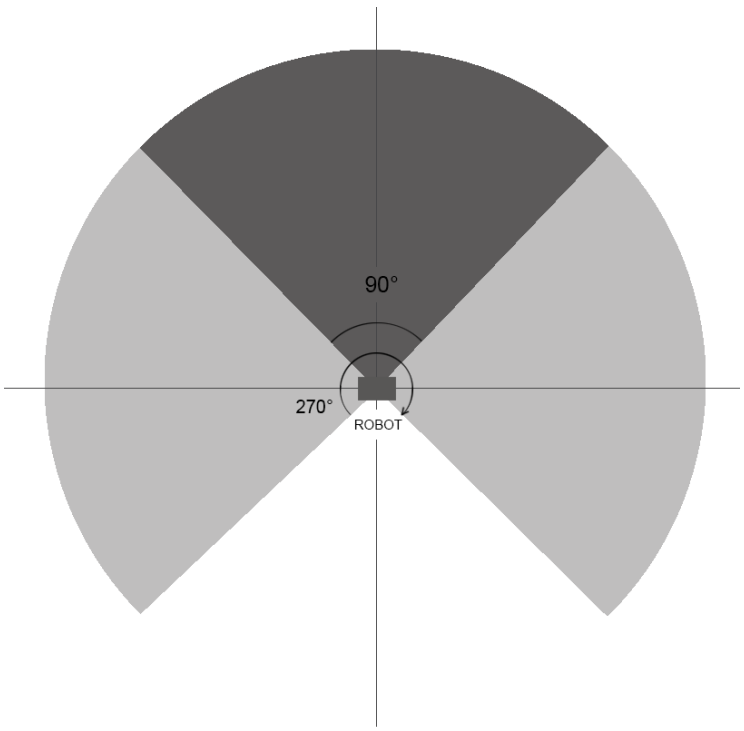

Figure 4: **Range of laser scanner.**

Note that for the purposes of this assignment, the robot is instructed to scan for obstacles within a 90°field-of-view. That is, obstacles further than 45°to the left or right of the centre of the robot are deemed as harmless and are ignored.

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

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). The robot does this by picking the closest 'safe' spot to the left of the obstacle, if possible, and driving 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.

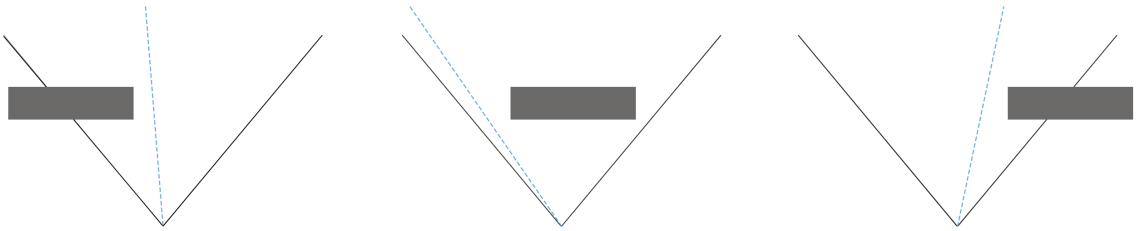

Figure 5: **Obstacle avoidance paths.**

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

Final Iteration

TODO!

1.4.2 Motor Control

TODO: motordata arduino send.

1.5 Open Source SICK TiM Driver

2 Results

3 Future Work and Improvements

4 Appendix

4.1 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
```