Implementation and testing of a path tracker for a full-scale Unmanned Ground Vehicle

Marcus Rådman

Computer Science and Engineering, master's level
2019

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
This project is the implementation and testing of a path tracker for a car-sized Unmanned Ground Vehicle. The vehicle, a Toyota Land Cruiser, was provided by SSC and had previously been modified for remote operation.

The developed path tracker uses a "follow the carrot" algorithm and has been written in C using the Robot Operating Software (ROS) framework and has been integrated into the vehicle's existing ROS powered software.

During the implementation, the Gazebo rigid body simulator was used to simulate a simplified vehicle. Integration with the real sensors was performed using a small-scale car, both indoors with the aid of a Vicon motion capture system and outdoors utilizing only sensors available to the full-size car. The small-scale tests showed promise, however when full-scale field tests were performed the results showed some problems and reasons for these are discussed.
This Master’s thesis is part of the work for my Master of Science degree in Computer Science and Engineering. The work for this thesis has taken place at the Control Engineering Group at Luleå University of Technology and in cooperation with Swedish Space Corporation (SSC).

During this project, I have learned a lot about what sort of non-obvious problems that can occur when trying to make different systems work together. It has also given me a reminder to never take the correct function of anything for granted; always test it first to make sure it behaves the way you expect.

Big thanks to my supervisors George Nikolakopoulos and Dariusz Kominiak for their guidance, aid, and patience, without their support the work for this thesis would have been much more difficult. I would also like to thank everyone else who have offered help, advice or support during the project. No one mentioned, no one forgotten.

Marcus Rådman
CONTENTS

CHAPTER 1 – INTRODUCTION 3

CHAPTER 2 – CURRENT STATE OF THE UGV 5
   2.1 Toyota Land Cruiser ................................. 5
   2.2 Hardware and sensors ................................ 5
   2.3 Software ............................................. 7
      2.3.1 ROS ........................................... 7
      2.3.2 Existing software .............................. 7
   2.4 Small-scale test vehicle ......................... 8

CHAPTER 3 – THE PATH TRACKING ALGORITHM 11
   3.1 Selection ............................................ 11
   3.2 Description ........................................ 12
   3.3 Limitations ........................................ 13

CHAPTER 4 – DEVELOPED SOFTWARE 15
   4.1 Overview .......................................... 15
   4.2 Issues ............................................. 17

CHAPTER 5 – SIMULATION 19

CHAPTER 6 – SMALL-SCALE TESTING 21
   6.1 The lab environment ............................. 21
   6.2 Indoor small scale-tests ....................... 21
   6.3 Outdoor small scale tests ..................... 22

CHAPTER 7 – FULL-SCALE FIELD TEST 25

CHAPTER 8 – DISCUSSION AND CONCLUSIONS 33

CHAPTER 9 – FUTURE WORK 35
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>UAV</td>
<td>Unmanned Areal Vehicle</td>
</tr>
<tr>
<td>UGV</td>
<td>Unmanned Ground Vehicle</td>
</tr>
<tr>
<td>ROS</td>
<td>Robot Operating System</td>
</tr>
<tr>
<td>URDF</td>
<td>Unified Robot Description Format</td>
</tr>
<tr>
<td>IMU</td>
<td>Inertial Measurement Unit</td>
</tr>
<tr>
<td>SLAM</td>
<td>Simultaneous localization and mapping</td>
</tr>
<tr>
<td>PWM</td>
<td>Pulse Width Modulation</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>PID controller</td>
<td>Proportional-Integral-Derivative controller</td>
</tr>
</tbody>
</table>
The area of autonomous self-driving cars is not a new one and research into the topic is decades old; starting with remote controlled vehicles in the 1920’s and ranging to the possibility of large-scale deployment of fully autonomous vehicles on public roads in the not too far distant future.

The interest in autonomous vehicles has many reasons. For example; an unmanned vehicle can be used to explore or work in environments deemed too dangerous to risk human life; be it the surface of another planet, rural areas plagued by forest fires, an unstable mine shaft, or a collapsed building.

There is also potential to improve safety in human-occupied vehicles. A large number of traffic accidents are due to human error and self-driving cars offer a possible way of reducing the accident rates.

There are currently multiple projects around the world developing self-driving cars, one of the more well known commercial projects is the one from Uber. When driven autonomously there is always a safety driver ready to take manual control if necessary. In 2016 Uber began to make it possible for their paying customers in select areas to hail one of these self-driving cars and be driven to their destination.[1]

Another high profile autonomous car project is Waymo, which began as the Google self-driving car project in 2009. The first vehicles were based on Toyota Prius and other cars have been modified for the project. Not all vehicles are based on existing cars however, the Firefly, a small prototype car with room for two passengers were built as a custom development vehicle and lacks any provision for a driver and is solely meant to be driven autonomously. Waymo claims to have had vehicles driving autonomously on public roads for a total of more than 5.6 million kilometers since the beginning.

In the case of Waymo, their vehicles carry a suite of sensors to give a 360° view around the vehicle. The sensor data is cross-referenced with the vehicles high detail 3D maps to determine the vehicle’s location on the road. These sensors include visual cameras, radar as well as short-, medium- and long-range LiDAR. In addition, the vehicles also have an audio sensor in order to hear the sirens of emergency vehicles, and a GPS supplement
the vehicle’s understanding of where it is. [2]

The UGV in this project has been developed as a low-cost autonomous search and rescue vehicle for rural environments with software using the Robot Operating Software (ROS) framework.

As a search and rescue vehicle, it must be able to operate in unknown environments relying only or primarily on onboard sensors to determine where it can and cannot go due to obstacles and terrain.

SLAM using visual cameras is being studied for the vehicle, the aim is for the UGV to localize itself and create a map of its surroundings. The map could be used by a path planner to find a route through the area; when a dynamic map is unavailable the path planner could use a pre-generated map and perform the planning offline.

There would however be little point in a path planner if the vehicle does not have a system to follow the paths the path planner generates. In this project the focus is on the implementation and testing of a basic path tracker, using pre-defined paths with no consideration for any obstacles.
Chapter 2

Current state of the UGV

2.1 Toyota Land Cruiser

The main target vehicle in this project is a Toyota Land Cruiser. The system could however be easily integrated into any car, provided the car was equipped with an automatic gearbox. The Land Cruiser was provided by SSC and had previously been modified and remotely operated at the Vidsel Test Range.

The vehicle is controlled with actuators installed to control the throttle and brake pedals as well as the steering wheel. The actuators are all electrical servomotors with an internal control loop and are controlled via PWM signals and can be seen in figure 2.2.

2.2 Hardware and sensors

The electronic hardware used on both the small test vehicle and the full-scale car consists of a Raspberry Pi single board computer as the main computational unit. The Raspberry Pi is connected with WiFi to a computer that gives it commands but all calculations take place on the Raspberry. If the connection to the computer is lost the vehicle will continue to follow the path at its current speed. This is an obvious safety issue and all tests of the path tracker were performed at low speed and with us inside the car to be able to stop is in case anything went wrong.

A "Mini Maestro 12" USB servo controller board is used to generate the PWM output signals to the servos.

Only two sensors were used in this project, with the purpose of determining the vehicles position and orientation. A GPS board to provide position and an IMU to provide rotational data.

The IMU, an Xsense MTi-1, is capable of measuring both linear and angular acceleration. Only the angular data is used, however, in order to determine the vehicle’s orientation.
Figure 2.1: The Toyota Land Cruiser UGV

Figure 2.2: Servos installed in the car
2.3 Software

The software is a ROS (Robot Operating System) application written in Python and C++ and allows a driver to directly command the steering wheel servo and command a velocity to the vehicle.

2.3.1 ROS

ROS stands for Robot Operating System, and despite the name, it is not actually an operating system, rather it is a framework to allow smaller programs to communicate with other programs and together form a larger application. The individual programs are called nodes and the communication can be both asynchronous and synchronous. Nodes can be written in Python or C++, and as all communication goes through ROS, there are no special provisions necessary to have both Python and C++ nodes work together.

In ROS, asynchronous communication is achieved through a ROS topic. A topic is, in essence, a subscription service that a node can subscribe to; when another node publishes a message to the topic, ROS distributes that message to all subscribers to that topic.

Synchronous communication is enabled through ROS services. When using a service the client node will then block and wait for the servicing node to return, similar to a regular function call.

All topics or services are of a specific type, containing certain data types. These topic types and service types are defined in separate files.

2.3.2 Existing software

A command-line interface in the ”Servo_mover” ROS node allows the driver to give a target speed to the speed controller and command the steering servo. The speed commands go through the ”Velocity_mixer” node to a node containing a PID controller to control the vehicle’s speed. The ”Velocity_mixer” node’s only purpose is to read the velocity setpoint from the driver and the current speed from the localization system and present both in a single message to the ”velocity_pid_controller” node containing the PID controller. The speed control signals from the PID controller and the brake and steering commands from the user interface is received by the ”Servo_listener” node, responsible for outputting the control signals to the servos. While the vehicle is not commanded to stand still the throttle is commanded by the PID controller to achieve the target velocity. The brake pedal is only pressed when the vehicle is commanded to stop.

From the sensor data, a position estimate is generated by using the ”robot_localization” package, a ROS package providing state estimation trough ether an extended Kalman filter or an unscented Kalman filter. The package also contains a node to transform satellite navigation data into a local reference frame for convenient use in the Kalman filters. In the software, the extended Kalman filter node is used and its output is presented in the ”/odometry/filtered/global” topic. In addition to being used by the speed controller,
the localization date is also used to plot the vehicle’s position. [5]

2.4 Small-scale test vehicle

To serve as a small scale test vehicle, an RC car was modified by removing the radio and control electronics and replacing them with the same sensor and computer hardware, running the same software, that is used on the full size vehicle. As is illustrated in figure 2.4.

It is powered with a 4 cell lithium ion battery to drive the main electric motor and a regulator that supplies 5V DC to the electronics and steering servo.
Figure 2.4: Difference between full scale and small scale vehicle
CHAPTER 3

The path tracking algorithm

The algorithm is an implementation of a Follow-the-Carrot scheme working by linear interpolation between waypoints, finding the closest point to the vehicle, following the path a pre-determined distance and then steering towards that point.

3.1 Selection

To select a path tracking algorithm information on commonly used algorithms was examined.

One work who actively compares different algorithms is Martin Lundgren in 2003, comparing Follow-the-Carrot and Pure Pursuit, both established and well known algorithms, with Vector Pursuit[6]. These algorithms were tested on a small differential steering robot following a variety of different paths. In these, Pure Pursuit performed the best in most situations with Vector Pursuit surpassing it in certain more extreme situations. Follow-the-Carrot performed worse than both in almost all of the tested circumstances but it is simpler to both implement and understand.

The simplicity of Follow-the-Carrot was appealing despite its drawbacks, but the relative ease of converting it a to a Pure Pursuit algorithm at a later time lessens the effect of implementing one and then change your mind. The main difference between them lies in determining how to steer once the carrot is found. Follow-the-Carrot tries to steer directly towards the carrot, Pure Pursuit on the other hand tries to adjust the turning radius such that the turn intersects the carrot point. With this in mind and cautious of any issues that might show up during implementation and testing, simplicity was deemed more important and the decision was to implement a Follow-the-Carrot algorithm with a possible future upgrade to using Pure Pursuit.
3.2 Description

The path is defined on start-up by a series of waypoints and linear interpolation between them. The path is provided to the software in a YAML file containing the boolean parameter "cyclic" and the list of waypoints. YAML is a data serialization language used for configuration files among other things.[7] If "cyclic" is true, the path will we closed with a line segment from the last waypoint to the first.

Other than the path, the input consists of the vehicle position in a local x- and y-coordinates, vehicle heading angle relative to the x-axis, and two more parameters, "segment_lookahead" and "carrot_lookahead".

When the vehicle position is updated the first step is to find the point on the path that is closest to the vehicles current position. In order to allow a path to intersect itself without the vehicle skipping part of the path and to allow the path to be cyclic and repeat itself over and over, only the previously closest segment and a limited number of those following it are considered when finding the closest segment. For this purpose, an internal state is maintained to keep track on which segment of the path the closest point was last situated on. Not considering the entire path also reduces the number of calculations necessary, especially in the case of paths consisting of a lot of waypoints. The number of segments is determined by the "segment_lookahead" parameter and for each segment to be considered the closest point on the line between the waypoints and the vehicle position is calculated.

To find the closest point for a segment, a vector $v_{seg}$ is created from the coordinates of the beginning waypoint and the end waypoint,

$$p_{wp_n}, p_{car} \in \mathbb{R}^2$$

$$v_{seg} = p_{wp_{n+1}} - p_{wp_n}, \quad v_{seg} \in \mathbb{R}^2$$

And a vector to the vehicle from the beginning waypoint

$$v_{car} = p_{car} - p_{wp_n}, \quad v_{car} \in \mathbb{R}^2$$

The closest point on the line spanned by $v_{seg}$ is obtained by an orthogonal projection of $v_{car}$ onto $v_{seg}$

$$v_{close} = \frac{v_{seg} \cdot v_{car}}{v_{seg} \cdot v_{seg}} v_{seg}, \quad v_{close} \in \mathbb{R}^2$$

As only points between the waypoints are of interest, if the projection falls outside of $v_{seg}$, the projection is clamped.

$$v_{close} = \begin{cases} 0, & \text{if } \frac{v_{seg} \cdot v_{car}}{v_{seg} \cdot v_{seg}} < 0 \\ v_{seg}, & \text{if } \frac{v_{seg} \cdot v_{car}}{v_{seg} \cdot v_{seg}} > 1, \\ \frac{v_{seg} \cdot v_{car}}{v_{seg} \cdot v_{seg}} v_{seg}, & \text{otherwise} \end{cases} \quad v_{close} \in \mathbb{R}^2$$
3.3  Limitations

![Diagram of path planning](image)

Figure 3.1: Illustration of how the output angle is determined

The position of this closest is obtained by

\[ \mathbf{p}_{\text{close}} = \mathbf{v}_{\text{close}} + \mathbf{v}_{\text{wpn}}, \quad \mathbf{p}_{\text{close}} \in \mathbb{R}^2 \]

The distance from the vehicle to the closest point of each segment is saved to be compared with the results from the other segments. The closest of these close points are selected as the overall closest point and the closest segment is saved for the next round.

From the closest point, if the distance to the next waypoint is longer than then the lookahead distance, the carrot point is chosen as the point "lookahead distance" away from the closest point in the direction of next waypoint. Otherwise, the distance to the waypoint is subtracted from the lookahead distance and this value is used as the temporary lookahead distance and the waypoint is used as a temporary close point in another attempt to find a carrot point. This is repeated recursively until a carrot point is found.

Once the carrot is found the angle to the carrot from the current heading of the vehicle is calculated and the angle is output as the desired steering angle.

In figure [3.1] the closest point \( \mathbf{p}_{\text{close}} \) had been found thorough the process described above, from \( \mathbf{p}_{\text{close}} \) the path is followed the pre-determined lookahead distance and the carrot \( \mathbf{p}_{\text{carrot}} \) is placed. The output from the algorithm is the angle \( \theta \).

3.3  Limitations

Due to the carrot point moving ahead of the vehicle and when nearing the waypoint the carrot will be on the next segment, the vehicle will start turning before reaching the waypoint and therefore try to cut any corners. For a vehicle to be able to exactly follow the path all the way to the waypoint before heading to the next without cutting or overshooting any corners, the vehicle would need a zero turning radius and be able to turn on the spot. In order for a vehicle with a non-zero turning radius pass exactly over
a corner waypoint the would either have to wait until passing over the waypoint before it starts turning towards the next and therefore overshoot the next segment, or turn away from the waypoint before reaching it, reverse the turn and star steering towards the next waypoint and pass over the current waypoint mid-turn. Starting the turn before reaching the waypoint is inherent to the follow the carrot algorithm and even a vehicle that is capable of turning on the spot will turn before the waypoint.
The software that has been developed is a path tracker in order to allow the vehicle to follow a pre-programmed path, and any modifications necessary for the path tracker to interact with the existing software.

4.1 Overview

The existing localization system is retained and the resulting estimation of the vehicle’s position and orientation is then used by the path tracker to determine where along the path it is and find a point to steer towards. The details of this algorithm have been described in chapter 3. The angle to that point is then passed on to the steering controller which is a simple multiplier and commands the steering servo.

The speed controller remains unmodified and is independent from the path tracker. Coupling the speed controller with the path tracker was considered to for example allow the vehicle to slow down when turning. However with the priority on just the path tracker; the coupling has been left to future work.

The command-line interface has been modified to be able to activate and deactivate the path tracker but retains the ability to set a target speed. When the path tracker is not in use, the input topic to the servo controller can be remapped and allow the interface steer the vehicle directly.

The path is provided to the path tracker through a YAML file, containing a Boolean and, a list of x- and y-coordinates. Two additional parameters are provided in the path launch file, segment_lookahead and carrot_lookahead.

Segment_lookahead defined how many segments the path tracker should consider when determining the closest point. segment_lookahead = 2 means the previously closest segment and the one following it will be searched. Segment_lookahead < 0 is a special case, instructing the path tracker to consider all segments.

A segment lookahead value of 1 would mean only the previous closest segment would
Figure 4.1: Structure of ROS node
be considered, resulting in the next would always be the previous one, preventing the path tracker to progress when reaching the end of that segment. The carrot_lookahead is how far ahead of the closest point, in meters, the carrot should be placed.

The command-line interface has been extended to also give enable/disable commands to the path tracker. An overall view of the ROS node structure can be seen in figure 4.1.

4.2 Issues

The Robot localization package used expects yaw data from the IMU as radians relative to the east. Because the IMU’s zero yaw is determined by its direction at startup (in its current state), it is necessary to set the "yaw offset" parameter. The simplest way I have found to get a good yaw offset is to perform a test drive along the desired local x-axis with the vehicle starting facing in the desired positive x-direction. First determine a rough estimate of the heading of the desired local x-axis, either through outside knowledge or by using a hand compass, it is sufficient to be within \( \pm \frac{\pi}{2} \) (\( \pm 90^\circ \)) of the correct heading in order for the result of arc-tangent in the next step to be in the correct range. By then manually diving along the desired x-axis it is possible to determine how to further adjust the yaw offset by taking the arc-tangent of the end position, \( \tan^{-1}\left(\frac{y}{x}\right) = \theta \), adjusting the yaw offset by this amount and then, always have the vehicle facing this direction when starting the software.

The "xsens_driver" node seems somewhat unstable and occasionally crashes, forcing any tests to be aborted and the software restarted.
To enable testing of the path tracker during the implementation of it without the problems and risks of using the real car, a simplified vehicle model was simulated in Gazebo.

Gazebo is a rigid body simulator able to be easily integrated with a ROS application through a variety of ready-made plugins. Gazebo can through plugins provide simulated sensor data to the ROS topic which normally receives data from the real sensor. Other plugins allow the software to control the robot in the gazebo world by applying force and torque to the joints or parts of the robot. By only replacing the interfaces with the real world with simulated ones, no major modification of the path tracker was required when moving to the small scale, and later the full-size vehicle.

The model itself is described in a URDF file. URDF is an XML format used in ROS to describe the physical elements of a robot. The elements, called links, can contain information on the links mass, inertial characteristics, collision geometry, and visual geometry. The collision and visual geometry does not have to be the same and can consist of primitive shapes, such as cuboids or cylinders, or custom meshes. Typically the collision geometry is a primitive, or a low polygon mesh, even if the visual is a more complex shape.

The car itself is a four-wheel vehicle with front wheel Ackerman steering. To simplify the model a tricycle with two rear wheels and one front wheel for steering was used. This works well as the Ackermann steering has both front wheels oriented around a common center of turn based on an ideal, and non-existent, wheel placed in the middle between the front wheels.

The model consists of a few simple shapes a cuboid as the main body of the car, a small cylinder to serve as the mounting point for the front wheel, and three larger cylinders as the wheels the small cylinder connected to the body with a revolution joint able to turn about the z-axis, the front wheel connected to the small cylinder with a revolution joint able to turn about the y-axis. rear wheels connected directly to the body with revolution joints able to turn about the y-axis.

When creating the model in URDF, Rviz was used to visualize the model and make sure
all links and joints connected and could move correctly. Rviz is a ROS 3D visualization tool that can be used to show positions, trajectories and URDF visualizations, among other things. This visualization can be seen in figure 5.1, the visual element of the vehicle body is intentionally undersized in order to see the wheels.

The model vehicle is controlled via a ROS plugin to gazebo that can apply torque to the wheel joints. The torque is applied by PID controllers to regulate the speed of the wheels, with all wheels having their own PID controller that is all set to the same setpoint and parameters. The front wheel mounting point is controlled by another PID controller to control the rotation and steer the vehicle. The steering PID controller listens to the "CarAngle" topic. The PID controllers for the speeds are listening to the "CarVelocity" topic directly from the Servo_mover node, bypassing the "Velocity_mixer" and "velocity_pid_controller" nodes.

The nodes for the IMU and GPS as well as the robot_localization nodes are not used, instead, position and orientation data comes directly from the gazebo plugin. This means the path tracker is working with the actual position instead of a more or less accurate estimation.
To test the path tracker once implemented the small-scale vehicle described in section 2.4 were used and tests were performed both indoors and outdoors. The indoor testing took place inside the robotics lab taking advantage of the lab’s motion capture system. Outdoor tests allowed the GPS to be used, completing the electronics of the full sized vehicle.

6.1 The lab environment

The lab is equipped with a Vicon Motion Capture System that can accurately determine the position of reflective balls and by attaching several balls to an object the objects position and orientation can be determined. The Vicon system in the lab has twenty VICON T40 infrared cameras and the system can determine the position of the reflective markers with an accuracy of 0.1mm inside the lab. The system does this by analyzing the images from all the cameras looking for the markers and comparing the images to be able to place each marker in a three-dimensional space.

In the lab environment the coordinate system is fixed between test runs as opposed to the outside tests where the position of the coordinate system origin in the real world depends on the starting position of the vehicle. The fixed coordinate system has the advantage of waypoints not moving between rounds even if the vehicles starting position is not the same.

6.2 Indoor small scale-tests

The paths are described in this text as figure eighth or a circle but, by necessity from being defined by a finite number of waypoints, are in fact polygons approximating these shapes. The paths have been defined by hand to have them without setting up a system to auto-generate them, the paths have therefore been given relatively few waypoints to
keep them manageable. The limited floor space inside the lab meant only relatively small paths could be used.

The Vicon system position data, containing both position and rotation, is fed to the path tracker in place of localization data from the GPS and IMU in a similar way to how gazebo returns the simulated position to the path tracker. The Vicon data does, however, go to a different ROS topic as opposed to the simulated data that goes to the same topic that would be used by the localization system and the path tracker is set to use the Vicon topic instead. The data goes from the Vicon computer that compiles the data from the cameras into position and rotation data to the Raspberry Pi on the vehicle over wi-fi via a laptop.

Initially, the path tracker used both position and orientation data from the Vicon system. This verified the electronic hardware and the path tracker worked in a real-life case; it also allowed initial parameters to the path tracker to be tested.

In addition to the parameters to the path tracker, the vehicle’s behavior is also dependant on speed. For example with a square path, at low speeds, the vehicle starts turning before reaching the corner and then follow the next edge until turning before the next corner. With the same settings but higher speeds, the vehicle would often start turning before reaching the corner but overshoot the next edge slightly before following the edge in diminishing oscillations until reaching the next corner.

The next step was to integrate the IMU data into the path tracker. The path tracker is still set to use the position from the Vicon topic but to use orientation data from the localization topic fed by the IMU. To begin with, the orientation produced by the localization package was rotated by 90 degrees from what was expected, but this was corrected by changing the yaw offset parameter in the localization packet configuration. Once this issue was fixed the vehicle behaved very similar to when both position and orientation came from the Vicon system.

6.3 Outdoor small scale tests

The outdoor tests were performed with the small-scale vehicle with the full set of hardware active and relying on the GPS for positioning.

The initial outdoor test paths were a line straight ahead and the vehicle would turn and drive in a completely different direction. To try and determine the cause of this the vehicle was picked up and moved in different directions with the software running. It was noted the positive x-axis was always in the same direction as opposed to being straight ahead of the vehicle on start-up, regardless of what direction the vehicle was facing. This was determined to be because the localization package assumes the direction from the sensors are relative to the eastward direction, modified by the yaw offset parameter, and the IMU gives its rotation relative to its starting orientation. In section 4.2 the solution used is described.

With the heading issue resolved another problem became apparent, the estimated posi-
tion would move a lot even with the vehicle being stationary. After attempts to improve the accuracy of the position estimate, the problem came down to the accuracy of the GPS module. Different positions of the GPS antenna were tested, including well away from the other electronics in case of interference. The system was also left running over 20 minutes to see if accuracy improved over time. These efforts failed and culminated in the replacement of the GPS module. The new GPS module is a NEO-M8N Evaluation board and this choice came down to it being immediately available and performing much better.

Figure 6.1: Small scale vehicle following a figure eight path

Figure 6.1 shows the plot of a test run with a figure eighth shaped path, a type of path that does intersect with itself. The trace shows the vehicle starting at the origin and following a figure eight smaller than what is given by the waypoint, the main reason being the vehicle is continuously cutting the corners of the programmed path. In the plot there are a few conspicuous straight lines in the trace, these are due to the vehicle momentarily losing the connection to the computer recording the data.
The full-scale test took place on an asphalt runway approximately 18 meters wide with a strip of gravel on either side. Figure 7.1 shows the UGV on the runway during one of the test runs. A number path files were prepared ahead of time, the test runs were managed from the car’s backseat and were performed at low speed with data recorded for later more detailed analysis.

In figure 7.2 the actuators have been installed and the path is set to a straight line. The vehicle did remain close to the desired line but with a noticeable oscillation. The
oscillation can be seen more clearly in figure [7.3] generated from the same data but with a different scale on the axis.

For this, test the lookahead distance was set to 5 meters. Another test with the same path but with the lookahead distance set to 10 meters, that test showed a larger oscillation than the previous one and due to time constraints, in all subsequent tests, the 5-meter lookahead distance was used.

![Image](image.png)

*Figure 7.2: Straight line, Lookahead distance 5m*

The next path was a rectangle 14 meters wide and 50 meters long and the trace of this test can be seen in figure [7.4]. The rectangle would have been wider if the paved had also been wider to accommodate it. The vehicle started turning as expected but overshot the long side of the rectangle, going out into the gravel beside the runway before turning back towards the path. It then reached the end of the rectangles long edge, overshot the short edge and made a too wide turn, heading off even further into the gravel on the other side of the runway before turning back. Again the vehicle reached the end of the long edge and performed a too wide turn into the gravel on the other side.
Next was the figure eight paths, of the largest pre-prepared figure eight path were tried it was apparent it was too small for the vehicle to have any chance to successfully follow it. Two figure-eight paths were modified to be bigger and they were also made proportionally longer since the runway length did not pose a limitation. Figure 7.5 shows the result of one of them. The path consisted of 6 waypoints and the shape was 10 meters wide and 80 meters long. The vehicle overshot the path but made the turn through the center and then back towards the tip but when reaching the far end of the eighth the car, similar to the rectangle test, made a too wide turn back, going off the runway during the turn. Once back on the runway the vehicle did not overshoot to the degree it went off the runway again on the opposite side but when it reached the tip of the path it again made a wide turn into the gravel strip beside the runway. This pattern was repeated every lap.

Another figure eight path, also with a width of 10 meters and 80 meters were also tested, but this one with only four waypoints, resembling a stylized hourglass. The results were similar to the 6 point path, with the car making a wide turn into the gravel strip when turning back at the far end of the path. The plot of this test can be seen in figure 7.6.

Finally, a rough serpentine-like path was prepared to see how the vehicle would handle a path where it would not have to make a tight 180-degree turn. The vehicle did overshoot the path and went off the runway, but not to the same degree as the previous tests.
Figure 7.4: 14m x 50m rectangle, Lookahead distance 5m, second attempt
Figure 7.5: 10mX80m figure eight, 6 points
Figure 7.6: 10mX80m figure eight, 4 points
Figure 7.7: rough serpentine
Discussion and conclusions

One problem is that some of the existing software is expecting data in non-standard units, particularly if angles should be given as radians or degrees. I decided to use ROS standard units and perform all calculations in radians, which also means I can utilize built-in functions for some calculations, and convert to degrees the last moment before sending data to the topics expecting degrees.

I probably should have taken the time to set up a way to auto-generate paths of different sizes for a few shapes. Creating a rectangular path with four waypoints is quick and easy but circular or figure eight paths with more points by hand could be a bit time-consuming, particularly if I made a mistake and had to go back and correct them. Finding that the path was the wrong size and had to be re-done happened more than once and while I am not certain I would have saved time overall by setting up the auto-generation, it would have made it more convenient.

It became apparent to me how important it is to have good odometry data, if you can not get a reasonably accurate estimation of the vehicles position and rotation, any path tracking or path planning will be of no use and could potentially do more harm than good as the vehicle veers of course. A lot of work went into tuning the localization system and getting good sensor data to feed it. Part of this was figuring out the yaw offset issue described in section 4.2. Another problem I had was the poor accuracy of the GPS module used at the time. Ultimately this was resolved by replacing the GPS module but during the process of trying to get more consistent data, I attempted to include acceleration for the IMU in the localization systems position estimate. I did not notice any improvements from doing that and in some cases the position ran out even further, despite the vehicle standing still. With the new GPS module giving me a more than acceptable position fix, I did not pursue the matter of using the acceleration further and left the localization system using only GPS for position.

The small scale vehicle performed significantly better than the full-scale vehicle and probably the main reason for this was the paths were significantly bigger relative to the minimum turning radius compared to the full-scale car.
The car, when driven by a human driver, is capable of making tighter turns than was demonstrated during the field test. Due to limitations on how much the steering wheel actuator could turn, the minimum turning radius was larger than expected, resulting in the vehicle being unable to make tight turns. As a result, most of the paths, believed to be large enough, were too small, not giving the vehicle enough room to settle on a segment before reaching the next. I had measured the turning radius of the car when driven by a human but it did not occur to me that the minimum turning radius could be that much bigger under computer control than under manual control. The larger turning radius also results in the vehicle not starting to turn in time to complete the turn and overshoot the next segment, as a result, a longer lookahead distance would have been required to compensate.

Another important reason the vehicle doesn’t steer as desired is the lack of an adequate steering controller. The steering signal goes from the path tracker, gets multiplied, and is fed straight to the servo without going through a software implemented feedback controller. A more advanced control scheme for the steering would have been desirable but for that to be possible, additional sensors to give the necessary feedback would have been needed. There is an internal feedback loop inside the servo but that is presumably only a P controller, and this controller’s parameters is out of the software’s reach. The resulting steady state error is likely the main culprit behind the oscillations seen, particularly in the straight line test.

Because the steering controller is internal to the servo and different servos are used in the car and the small scale vehicle, the vehicles essentially have different controllers, neither of the feedback loops can be affected by the software. This certainly affects the differences in performance between the two vehicles. Ideally both vehicles would have more advanced controllers specifically tuned to the vehicles.

As has been previously stated the paths were generally too small for the car to handle, however, the size of the test site itself placed constraints on the size of the paths. In retrospect, some paths could nonetheless have been improved. In particular, the rectangle path shown in figure 7.4 could have been made longer to give the vehicle more distance to settle around the long edges. It would not have been possible to make it any wider however as the rectangle already took up most of the runway’s width.

Overall I would conclude the path tracker system, as currently implemented, shows promise but is not yet ready for use in the full-size UGV and further tests and tuning would be necessary as well as general development of the other systems.
The is a lot of work remaining before the vehicle would really be capable of autonomous driving. To begin with, the steering would need to be improved to be able to make tighter turns or the vehicle might be limited to relatively open areas where large turns are possible.

In order to make the vehicle capable of autonomously finding its way through a set of waypoints while avoiding obstacles, the vehicle would first need to be equipped with sensors to detect any obstacles and then have a path planner to feed the path tracker in order to avoid the obstacle. The path tracker should also become coupled with the speed controller in some way to allow the vehicle to drive at higher speeds when driving straight and slow down when turning without additional manual input from an operator to slow down and speed up.

A convenient but not necessary improvement would be a system to automatically determine the starting orientation of the vehicle to prevent the need to set yaw offset manually. This would simplify the setup and save time during any future tests.
REFERENCES


