ARC Findings and Conclusions

CS Capstone - CS 463

Oregon State University

Spring 2017

Cierra Shawe, Tao Chen, Daniel Stoyer

June 10, 2017

Abstract

Group 44's *ARC Findings and Conclusions* provides an overview of the Autonomous RC (ARC) project, the methods used to conduct research and implement software, findings of the research, and a synthesis of the findings. ARC is designed to used commodity hardware and uses the Robot Operating System. It was determined that with more time and resources, it is possible to create an autonomous RC car.

CONTENTS

1 Introduction

The purpose of ARC is to determine if it is possible to create a small-scale autonomous hardware and software implementation that is able to be retrofitted onto a Remote Controlled (RC) car. The goal of ARC is to have a vehicle capable of navigating from point A to point B while implementing obstacle avoidance. The vehicle should also be able to parallel park itself when told where the final location of the vehicle should be. ARC is also designed to cost under $4,000 to make the platform available for education, hobby, and commercial purposes. The components within the projects are also designed to require minimal fabrication and electrical experience. Obstacle avoidance and the low cost are what differentiate ARC from other autonomous vehicle projects, as implementations such as Georgia Tech's AutoRally platform cost upwards of $40,000 to build. This makes ARC an order of magnitude cheaper than something such as the AutoRally platform. By providing access to a low cost autonomous platform, ARC would provide a way to test different autonomous features without having to use a full scale vehicle, lowing development costs and making autonomous vehicles more accessible to a wider range of users. The ARC software platform uses the Robot Operating System (ROS) in order to provide an easy interface for users to contribute to the project. ROS is a standard within robotics and it provides users the ability to customize the packages being used with minimal coding experience.

2 Methods

2.1 Trial and Error

We used the trial-and-error method to develop our software, because we had technically unlimited options. We could write our own software from scratch or there was a limited number of packages that were available from the internet. And we did both. We explored multiple open-source software, integrated with our own code. For instance, to develop the navigation functionality, we tried building another layer on top of the AutoRally project that will generate new waypoints to the "WaypointFollower". We did not choose to go this route because it was difficult for us to integrate obstacle avoidance to the layer, which would require a lot more time and resources. On the other hand, we discovered an open-source software that would do both navigation and obstacle avoidance, and issue commands to the control modules directly. However, we did not have too many options for some of the parts. This brings us to adaptation.

3 Adaptation

In the real world, engineers are required to work around the hardware that they were given. After an unsuccessful attempt to use an autopilot to handle the motor control and sensors, we were required to implement the functionality of the autopilot using using the GPS, IMU, Leddar, and Lidar that were provided. All of the components had ROS libraries that we were able to utilize, which made them easier to use; however, the GPS was not as accurate as we had hoped.

Custom component housings needed to be designed for all the hardware for items such as the GPS, IMU, Raspberry Pi, power converters, and motor control board.

ROS was a well-known platform for building robots. We did not have other options but to use it. Luckily, it worked very well; however, the learning curve was very steep at first. So it took us some time to learn how to use it.

3.1 Iterative/Modular Development

Using our requirements as a guide, we focused on getting every component to work individually before moving onto the next requirement. This allowed us to ensure that everything was able to work individually before attempting to integrate everything together.

4 Findings

4.1 Image Analysis

4.1.1 Fast Image Processing

We required image processing to be at least 15 frames per second (fps) for the vehicle to move at the speeds we desired. We found that image processing using stereo vision was too computationally expensive to run on our hardware. Much more expensive, super-computer-level hardware would be required to make image processing of stereo vision attain 15 or more fps.

4.1.2 Depth Finding

Our requirements to detect objects at least 6 feet away and display the objects in a GUI was met using a combination of a 360 degree Lidar unit and a front-facing Leddar unit. The LeddarTech M16 unit detects objects up to 50 meters in front of the car with a precision of 5cm at 50m. The RPLiDAR unit detects objects at 360 degrees up to 6 meters away with a precision of 1cm at 6m. Objects are displayed with the RVIZ visualization package.

4.1.3 Object Height Detection

We required our sensors to detect objects of 8 inches or more in height, from ground level. We have achieved that requirement using the front-facing LeddarTech M-16 which can detect a four inch object from six feet away from its position on the car. This exceeded our required object detection height of 12 inches.

4.2 Sensors

The main reason for needing GPS and IMU data is to determine the car's position in the world, termed localization. Localization can be achieved by using a combination of odometry, GPS, and IMU data, it can also be achieved with odometry. We wanted to omit odometry because retro-fitting a RC car with an odometry encoder is time-consuming and very technically/mechanically difficult. We discovered that achieving localization accurate enough for autonomous navigation requires GPS and IMU hardware much more precise than that available to us. Moreover, such hardware is quite expensive and would increase our costs substantially. Therefore, we see a needed trade off: localization can be achieved using a combination of lower-precision GPS and IMU hardware with an odometry encoder retro-fitted onto the RC vehicle. The advantage of this configuration is that it is rather inexpensive. The disadvantage is that retro-fitting the encoder is very time-consuming and requires mechanical expertise. On the other hand,localization can be achieved using only GPS and IMU hardware. The advantage is that you avoid retro-fitting an odometry encoder, the disadvantage is that such hardware is very expensive.

4.2.1 GPS Data Collection and Processing

Our requirements specified GPS accuracy to 10 meters outdoors in an open space with no trees within 50 meters. Our GPS unit meets the requirements, however we found that localization based off of GPS data requires much more accurate (and more expensive) equipment.

4.2.2 IMU Data Collection and Processing

IMU data was required to be sent to the main computer at a rate of at least 20 measurements per second and visualized a some sort of GUI. Our research revealed that it is standard to transmit such data at rates around 10 Hz, or 10 packets a second. We are able to send IMU data using ROS at 10 Hz, which is sufficient for our application. We have IMU data visualized through RVIZ and CLI.

4.3 Navigation

4.3.1 Motor Control

We required at least 75% accuracy when going forward and backward. We learned that odometer encoders were necessary to determine the accuracy of the motor control. Since we do not have odometer encoders, we are not able to determine motor control accuracy. We are able to visually confirm forward and backward operation of the motors.

4.3.2 Servo Control

We required a 25% margin of error with respect to the commands sent by the main computer at any instance. We are able to control servos on command, however we found that determining the exact margin of error of physical performance with respect to commands given requires more research with the physical hardware.

4.3.3 Probabilistic Analysis for Motion

We required that our filtered instance location estimation be of higher accuracy than our unfiltered instance location. We were unable to determine instance location accuracy on our physical car. Further research, implementing software state estimation on the hardware, will be needed to determine this requirement.

4.3.4 Motion Model

The commands sent by the computer were required to adapt to the vehicle's configurations. We were to confirm the correctness of the motion model on the vehicle through observation. As our vehicle is not currently in motion, we are unable to determine whether the motion model is working correctly for the physical car. The motion model does work within the simulation with our general car object.

4.3.5 Global Path Planning

We required a point-to-point global path via setting way-points. Our initial concept was to set multiple way-points, but it became clear that our global path needed to be a straight line, local path planning would handle obstacle avoidance and bring the car back on line with the global path. We currently have this functionality working in simulation only.

4.3.6 Local Path Planning

Our requirements for path planning were to generate a path of way-points that best fit avoiding local obstacles bringing the car back to the global path. We were able to achieve local path planning in simulation. Further time is needed to implement local path planning on the physical platform.

4.3.7 Obstacle avoidance

The requirements for obstacle avoidance were to avoid all obstacles the image analysis system could detect. Within the simulation, the car is able to avoid all obstacles, unless it senses that there is not enough room to reasonably get itself unstuck, in which case it will remain in the same position. The obstacle avoidance running in simulation should apply to the car running in real life, as the simulation accounts for the use of our LiDAR and Leddar sensors.

4.3.8 Parallel Parking

We required the vehicle to parallel park itself when given a open spot. We achieved this requirement in simulation being able to have the car park into a spot that is 2.5 times it's length. This is done by selecting the point, and direction, in which the user would like the car to end up. The smaller the space, the longer it will take to be in the proper position.

4.4 Hardware Mounting

Hardware mounting design is complete with CAD models. The mounts on the vehicle are currently in the prototype phase, we are waiting on facilities/resources to make 3D prints of the mounting models.

4.4.1 Minimize Port and Hardware Exposure

All ports and sensitive hardware needed to be isolated from the elements. We have a prototype body constructed and have all ports and points of exposure sealed with electrical and duct tape.

4.4.2 Secure Hardware Mounting

Hardware was required to be mounted securely in the event of collision or rollover. We currently have hardware mounted with zip ties and screws. As our mounting system is still in the prototype stage, it cannot withstand collision or rollover. We do have mounting hardware design in CAD software, but did not have time to have them 3D printed.

4.5 Communications

Radios were required for transmitting telemetry and basic commands from and to the vehicle. During research/testing we did have successful radio communication. However, once we abandoned the PXFMini autopilot we lost the API libraries that enabled our radios to function. More time and research is needed to build a software framework for sending and receiving radio data, presumably using MAVRos, a MavLink library for ROS. Further research is needed to determine how to implement an interface with the radios.

4.5.1 Telemetry, Vehicle Control, Emergency Stop

We required telemetry to be radioed from the car to a ground station. This was initially accomplished but is currently unmet, as explained above. Logically, vehicle control and emergency stop are not enabled.

4.6 User Interface

4.6.1 Graphical User Interface (GUI)

We required a GUI to see the current state of the vehicle and the sensors perceptions of its surroundings. The GUI was to also allow us to interact with the car and issue commands, such as way-point destinations. We are using RVIZ as our GUI, it shows us the state of the vehicle, the sensor data of the surroundings, and allows us to set way-point destinations.

4.6.2 Command Line Interface

A CLI was required to be able to enter way-points, and start and stop the vehicle. We did not meet this requirement. Our research into ROS tells us that such functionality can be written in either Python or C++, but more research and time is needed to implement the interface.

4.6.3 Way-Point Selection

We required way-point selection via either CLI, GUI, or both. We achieved way-point selection via GUI and the RVIZ software package. Vehicle Information We required a GUI to see all information regarding the vehicle. We are using RVIZ to accomplish this requirement.

5 Performance Requirements

5.1 Support of one user and setting 5 way-points

This is fulfilled using RVIZ, only one way-point is set, which meets the requirement of the global path. Capable of driving 10 mph max in straight line. We do not have tested information on this requirement. Our estimates from simulation suggest that the vehicle will be able to achieve 10 mph when travelling straight.

5.2 Capable of navigating indoor environment of 500 sqft.

It is undetermined whether the physical car will be able to navigate a room. From tests conducted on the vision system, the walls must not be made of glass.

5.3 Capable of navigating outdoor environment without space constraints.

The system is able to operate without constraints within our simulation.

5.4 Capable of parallel parking in under a minute.

Within our simulation, the car is able to parallel park within a space that it at least 2.5 times its length, in under a minute.

6 Design Constraints

6.1 Radio Communication within 2 km.

We were not able to test the radios, however the radio manufacturer (3DM) claims the radio range is up to 15km.

6.2 Autonomous navigation at 3 mph.

Our research through simulation suggests that this is very achievable. Further research is needed to implement the navigation stack on the physical platform.

6.3 Hardware Mounting

We needed to have mounting surfaces fixed to a pre-existing RC vehicle chassis. We have prototype mounting surfaces fixed to the vehicle. We have designed mounting parts using CAD models which need only be printed out with a 3D printer.

7 Conclusion

After close to 7 months of hard work, we can conclude that it is feasible to use only commodity hardware to build a platform that can autonomously drive a RC vehicle. As a research project, we hate to say that we have come to our conclusion without a working prototype; however, we made a list of things that we still need to implement based on what we have currently to make the platform work, and reasons why we cannot get to them. So far, we have had the software working on simulation. The software consists of 4 parts:

1) Descriptions of the vehicle
    a) Dimensions
    b) Max velocity, Max acceleration, Max steering angle

   c) Etc...

2) Description of the world
   a) An empty space where the vehicle always starts at the center of the world.
   b) Transformation of the vehicle to the world
   c) Max velocity, Max acceleration, Max steering angle
   d) 200 meters * 200 meters
3) Navigation Stack
   a) Path planning
   b) Obstacle avoidance
   c) Command issuing
4) Control
   a) Translation from commands issued by the navigation stack to the right values to control the motor and servos

We have everything else ready except 3a. "Transformation of the vehicle to the world" means the estimated location of the vehicle with respect to the world. Ex, the car always starts at the center of the world, which means the initial coordinate of the vehicle will always be (0, 0). Assuming a normal x-y coordinate system where y points to north and x points to east, if the vehicle move north for 10 meters, the location of the vehicle will become (0, 10). In simulation, the transformation is perfect because it is given by the simulation platform Stage. When we leave the simulation, and need to gather location information from sensors, it introduces huge uncertainties to the transformation. We only have one GPS unit and one IMU unit, which are the exactly opposite of "abundant". We believe if we have a system that can produce accurate and reliable location estimation, our platform will work. This brings us to "commodity hardware". We would like to elaborate on "commodity hardware". We have stuck with our project proposal where it limits us to only use commodity hardware. So far, every component we have used can be bought easily online. Prices may vary depending on the websites. We believe 2,000 - 3,000 US dollars will be sufficient (including buying the RC car) to build such a platform from scratch. As a research group, we did not realize that the research budget was different than the actual cost of the platform. The 2,000 - 3,000 US dollars we mentioned above was the estimated total cost to build the platform. But we needed more than that for the research, because we had to experiment with different components to get the best performance, without paying too much, which we did not do. We did not really have a research budget. Time is another reason why we could not build a working prototype. We know this is cliché, but it's really a factor. We got caught on some hardware that we found out later was useless to our project. We spent almost a term and a half on learning the ROS system and the AutoRally project. Also, we recently just realized that wheel encoders were highly recommended for doing location estimate for car like robot. We knew that wheel encoders were an option; however, we thought GPS and IMU were sufficient to give us accurate estimate, which was not true. Things like this delay our progress tremendously. We feel disappointed that we won't have the car ready for demo; however, we are confident our conclusion holds true.