Raspberry PiCar Object Detection, Tracking and Obstacle Avoidance

Sept. 3rd. - Dec. 9th. 2018

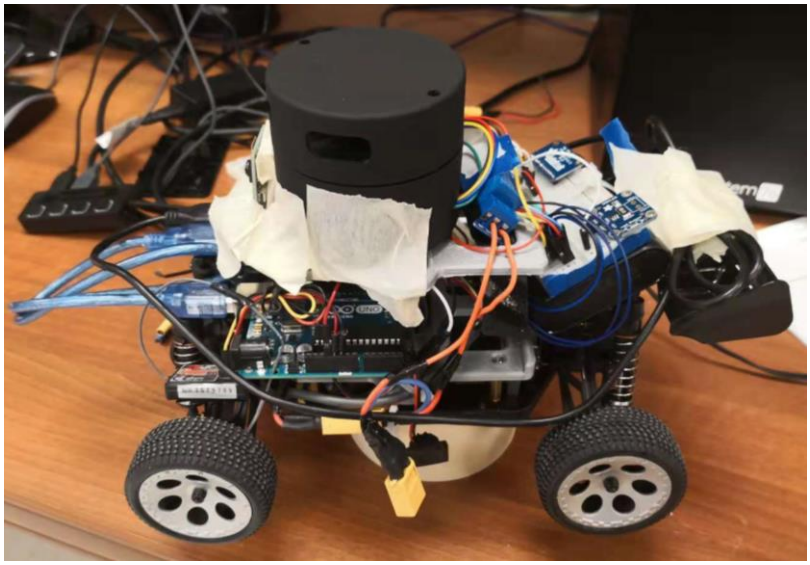
ESE 498 Capstone Design Project Final Report
Submitted to Professors Trobaugh and Feher and the Department of Electrical and Systems Engineering

Group Members
Amelia Ma, Electrical Engineering, B.S., amelia.ma@wustl.edu
Chufan Chen, Electrical Engineering, M.S. and B.S., chen.chufan@wustl.edu

Project Advisor
Xuan 'Silvia' Zhang, Ph.D, xuan.zhang@wustl.edu

Abstract

Object tracking and obstacle avoidance are two key features for a robot with mechanical movability and visual detection functionality. They are associated with many hot application fields such as path followers and self-driving cars. With the big picture in mind, we chose to implement those basic functions on the PiCar, a Raspberry Pi powered and wheeled robotic car, as a research project. By implementing the obstacle avoidance and the object tracking features onto the car, we aim to create powerful and practical algorithms that enable the car to follow a certain object wisely without crashing into any obstacles.



1. Introduction

1.1. Background Information

1.1.1. The Big Picture

The computer vision algorithms are evolving rapidly these days. The ability of "seeing" things is becoming a huge plus, if not an essential need, for a moving autonomous system from a self-driving car on the road to a small, personalized robot. In order to establish this "eyesight", one of the core algorithms needed for an autonomous machine is an object tracking algorithm, one that identifies and tracks objects within the machine's frame of vision. Another is an obstacle avoidance algorithm which guarantees that the machine safely avoids any obstacle on its way when driving forward to its destination. To design a machine with vision we are essentially designing these two features, obstacle avoidance and object tracking, which enable our robot to identify and follow a certain object without collision.

1.1.2. PiCar Platform

We decided to work on the existing PiCar platform. The PiCar is a "multi-purpose, robotic, lab-scale, open-source, wheeled, autonomous research platform" [9], a miniature four-wheel car powered by a Raspberry Pi board and an Arduino board. The Pi is a microcomputer that supports self-written, complex algorithms, and it communicates with the Arduino board which controls the physical movements of the car. The technical details of the existing platform can be found in the PiCar documentation webpage.

The car has an IMU sensor which is helpful for determining the location and the momentum of the car, and a Raspberry Pi Camera which is capable of taking pictures and videos of the surrounding environment. Additionally, a YD Lidar F4 was newly acquired (prior to the start of the project) to provide more accurate data on the distance of the surroundings.

We used the Lidar and the Camera for our project. Some metrics are listed below:

Lidar:
Scanning speed: 8 Hz (480 rpm)
Angle: 360 degrees with 0.5 degree resolution
Distance: maximum 12 meters with <1% error

Camera:
Resolution: 8-megapixel
Maximum image transfer rate: 30 fps (1080p) / 60 fps (720p)

1.2. Problem Statement

The PiCar was set-up and ready to run physically, but not many sophisticated algorithms were implemented yet. Things that already existed include the Arduino code for basic car movement control, the interface protocols between the Pi and the Arduino, and the interface between the Pi and the sensors. However, there was a lack of algorithms that could actively use the data collected. The PiCar had an object tracking algorithm that was functional only when tracking an object with strong contrast like a black and white chessboard, and it also didn't have a much compelling algorithm for obstacle avoidance. More importantly, its tracking and detection features were solely based on data from the PiCar camera, which is sensitive to the environment, and could be inaccurate at some time.

Our goal was to implement a better obstacle avoidance algorithm using data from the newly acquired Lidar, and to implement a camera-based object tracking feature to track a wider range of objects. This way, the car would be able to follow the selected object with better performance, and without collision.

1.3. Aims and Objectives

1.3.1. Obstacle Avoidance

Implement a working obstacle avoidance algorithm based on the Lidar input data that can detect obstacles in front of the car, and turn the car around to avoid crashing. The obstacles with any shape can be either stationary or moving at a speed lower than the car itself.

1.3.2. Object Tracking

Implement a working object tracking algorithm based on the Camera input data that can track a user-selected object of any shape. The object being tracked can move at a speed similar to the car's running speed.

2. Methods / Technical Approaches

Figure 1 illustrates the basics of our design:

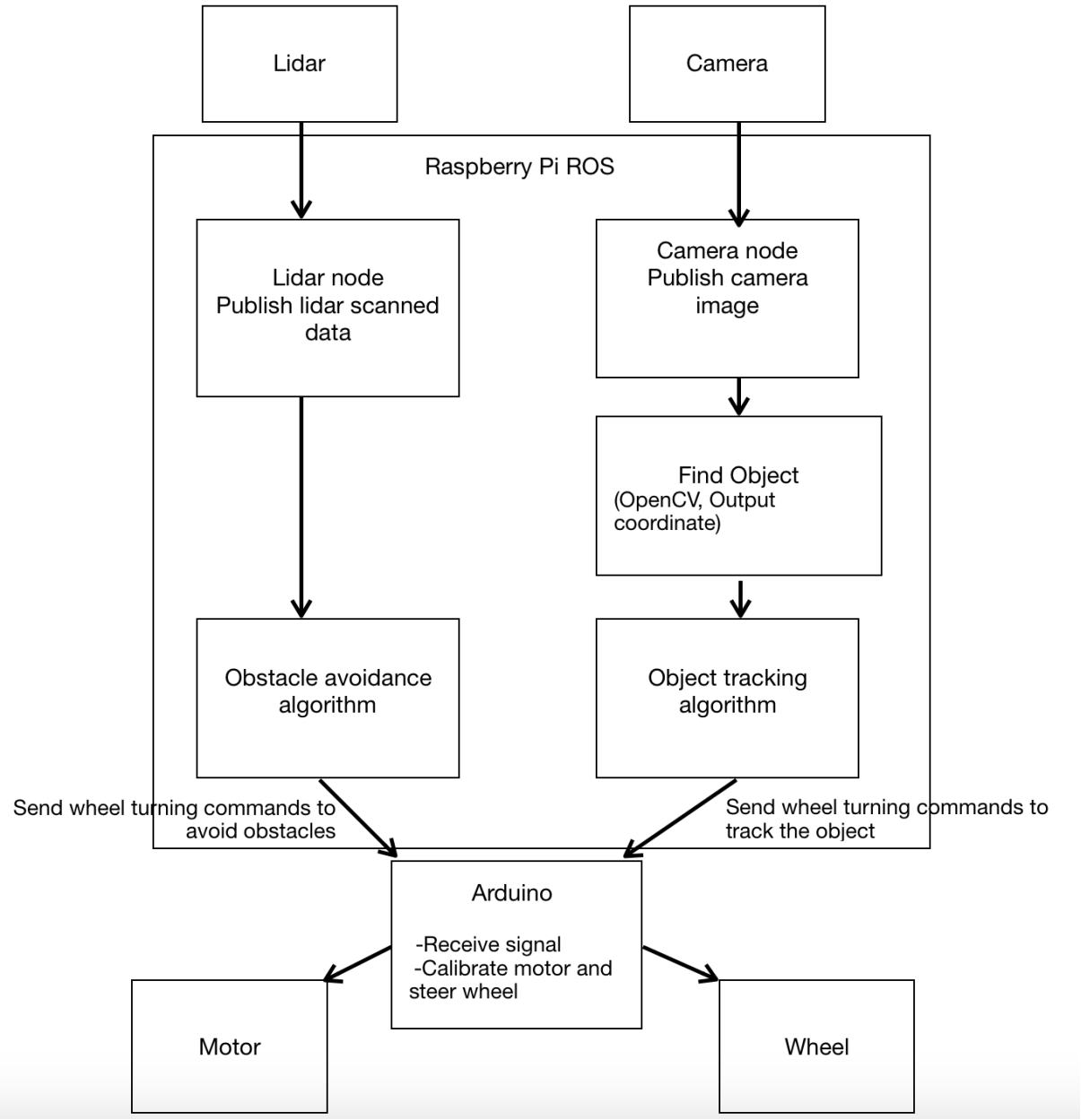


Figure 1: Diagram of the Design

Our design makes use of two different micro-computer boards: 1. The Raspberry Pi which takes in the sensor data, processes the data and sends out the car movement control signals, 2. The Arduino board which takes in the signals from the Pi and controls the physical parts of the car. The two boards each run a different set of algorithms and a communication channel in between is needed.

The design can be separated into four parts: **Data Acquiring, Data Processing, Pi-Arduino Communication,** and **Car Movement Control**.

Data Acquiring includes taking in the raw input data from the Lidar and the Camera, and pre-processing them to deliver the valid data that our algorithms rely on. The Lidar scans and

detects the distance of surrounding objects, and the data is sent in angle-distance pairs to the Raspberry Pi. The Camera takes pictures of its surroundings and the data comes into the Pi as pixels. *Data Processing* includes the obstacle avoidance and object tracking algorithms developed by ourselves, using the Lidar and the Camera data respectively. *Pi-Arduino Communication* includes sending control signals calculated by the two algorithms running on the Pi to the Arduino board. Both algorithms output signals through the Pi-Arduino Serial Communication channel. Finally, *Car Movement Control* includes the Arduino board controlling the physical parts of the car to make it actually move. The Arduino program will use the signals sent from the Pi to run the motor under a PID control algorithm, and to turn the car wheels to different angles.

Data acquiring and processing are both done on Raspberry Pi using ROS (Robotics Operating System), a very powerful platform for Robot developers. It is a separate operating system that could be easily installed on any existing systems, and it provides a collection of tools and libraries that could be utilized by the developers to better achieve their design goals. The details about using ROS will be covered shortly.

2.1. Data Acquiring

This section provides a brief introduction of ROS usage, split in three parts: **ROS Installation, Using ROS on YD Lidar,** and **Using ROS on Pi Camera**. The step-by-step documentation can be found on our project webpage (please refer to the link under the Appendix).

2.1.1. ROS Installation

The Robotic Operating System needs to be installed first to set up the environment for running the Lidar and the Camera. There are different ROS versions, and we chose to install the Kinetic version because it's relatively new and has stronger supports for various functionalities. We also chose to install ROS on Raspbian, the Raspberry Pi Operating System, because we wanted to keep aligned with the previous programs written for the PiCar, which were all running on Raspbian. ROS could also be installed on other platforms (for example, Ubuntu). The ROS wiki page for installing ROS [6] was a great resource.

Like ROS, Raspbian also has different versions, and the installation procedures vary for each version. Our Raspbian version was Stretch.

Upon installation, ROS has two sets of packages to choose from: a complete set with all ROS packages (tools and libraries), and a reduced set with the most essential ones. Generally, installing the reduced set is a common approach because it saves space and time. However, both the Lidar and the Camera depend on some packages that are not included in the common

set. It is possible to manually add all other packages required and create a customized installation set, but there are many of them, and dealing with a lot of dependencies can be tricky. Thus, we decided to simply install the “desktop” set which contains all packages.

Because the memory space on our Pi was not enough for ROS installation, we also added a swap file to allocate more memory for the process. The swap space was only needed for installation, not for running algorithms on ROS.

2.1.2. Using ROS on YD Lidar

Installing YD Lidar ROS Package

The Lidar comes with an open-source ROS package of nodes. ROS nodes are similar to computer processes: they are blocks of code running on the platform with certain features. One effective way to communicate between the ROS nodes is to make them Publishers and Subscribers, as what the Lidar source code does. A Publisher node, as the name suggests, publishes the data on the ROS platform with a unique identifier (called “topic” in ROS), and every Subscriber node that “listens” to that topic could receive the data. It is a one-to-many system: multiple Subscribers could subscribe to the same Publisher, as long as the specified topics match.

The YD Lidar package contains: 1. a ROS Publisher that takes in the Lidar data scanned, processes the data and “broadcasts” within the system, 2. a ROS Subscriber that “listens” to the Lidar data and feeds that into user-specified algorithms.

The thing sent by the Publisher and caught by the Subscriber is of the type “message”. It is like an object that has different “fields” that can be assigned by the Publisher and accessed by the Subscriber. In Lidar, the message type that the Publisher and the Subscriber used to communicate is *LaserScan* [18]. It is a common message type that applies to many devices that are scanning in data, and is ideal for Lidar data transfers.

The YD Lidar package also contains launch files. A launch file is not absolutely necessary for running the ROS node, but it is used to supply the parameters that need to be passed into the node. It specifies the package, the node, the Publisher/Subscriber communication topic, and the different parameter types and values. Whenever a launch file is present, the node needs to be launched via the launch file.

Because the YD LiDar package is a user-defined package not officially released by ROS, it needs to be installed separately.

Filtering Lidar Data

As discussed before, the YD Lidar uses *LaserScan* message type and outputs data in [angle, distance] pairs. The *LaserScan* data originally taken in is the raw data, which is really noisy. Thus, we needed to apply filters to remove the noise, which could make our obstacle detection algorithm more accurate.

There are several built-in *LaserScan* filters in ROS that specifically apply to data in *LaserScan* message type. Their source code could be found in the *laser_filters* ROS package [8] that builds upon the filters package.

One way to implement the filters is to use a scan-to-scan filter node. The node should be put in the middle of the Publisher (which publishes the raw scanned data) and the Subscriber (which receives the data). That way the Subscriber will receive the filtered data. The Subscriber should subscribe to the new topic of the filter node, instead of the Publisher topic.

Figure 2.1 and 2.2 illustrate how the scan-to-scan filter node works:

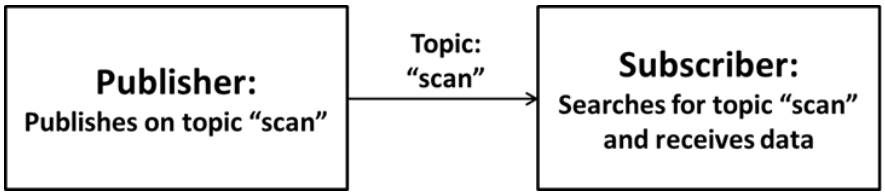


Figure 2.1: Before Adding the Node

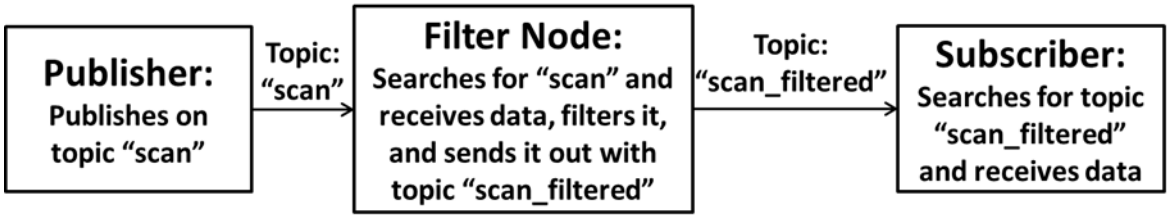


Figure 2.2: After Adding the Node

We picked the Median filter [4] provided in the package to filter our LaserScan data. The Median filter is a common type of filter which works by taking in the current measurement, finding the median of the current and a user defined number of previous values, and using the median as the current value.

The most important parameter for the Median filter is the number of observations, which indicates the number of values that the filter will choose the median value from. Bigger number of observations will generally lead to less noise but longer "reaction" time (the older values will "mask" the new values coming in, and the change in data will not be reflected until many cycles later). There is a tradeoff between the noise level and the reaction time, and we needed to find the optimal number of observations to balance the two. We set the number of observations for our Median filter to be 13, to reduce the noise to an acceptable level. For

the delay introduced by the filter, any change in the current incoming data will be reflected in the data array after six to seven observations, which is acceptable given the Lidar's scanning frequency. The Median filter does not perform heavy computations, and thus the overall delay will not influence the incoming data rate to an extent that needs to be taken into consideration.

2.1.3. Using ROS on Camera

We planned to use the built-in object detection functions from OpenCV to detect the object being tracked, and to provide the coordinates of the object for our tracking algorithm. The version of OpenCV on ROS Kinetic is OpenCV3.

Getting Image from the Pi Camera

The Raspberry Pi Camera takes in a 2D video stream of the surroundings, and each frame of the stream is composed of pixels. To obtain the raw image data, we used *raspicam_node*, which stands for Raspberry Pi Camera node [14]. It is a ROS package we downloaded and installed from the website. The node reads in the Camera image data and publishes it with the message type *sensor_msgs/Image* [17], which represents a 2D image and is ideal for Camera data transfers.

However, the message type *sensor_msgs/Image* is a ROS image format. If the user wants to use the image with OpenCV, there needs to be a "translator" between the ROS Image type and the OpenCV-compatible Image type. The package *cv_bridge* [12] is able to take the ROS image messages and convert them into OpenCV cv::Mat format which allows OpenCV to read and analyze them. *Cv_bridge* is also able to convert OpenCV image format back to ROS format.

Figure 3 below briefly demonstrates the process:

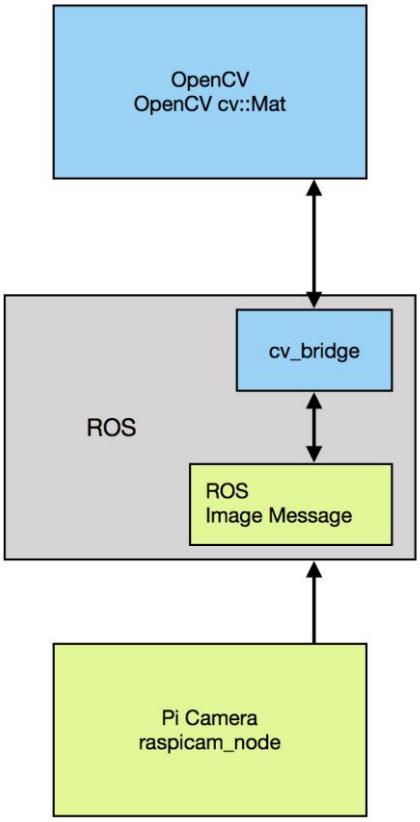


Figure 3: ROS and OpenCV Image Conversion

There are different resolutions to choose when launching *raspicam_node*. We chose to use resolution 1280x720, the lowest provided in the package, to allow quick and effective image data transfers.

Note that by default, the node only publishes the image in a compressed format (message type *sensor_msgs/CompressedImage*) [16]. Output of the raw camera image can be enabled by passing in command line arguments.

Viewing the Image

To see the captured image on the screen in real time, there needs to be another ROS package that displays the image, called *image_view* [11].

After the package is installed, launch the *raspicam_node*, and then run the *image_view* node to display the raw image on the screen. (This assumes that raw image publishing is enabled in *raspicam_node*.)

Locating Object Being Tracked

There are many useful ROS packages that use OpenCV to post-process the images captured by the Camera. Our goal was to track a certain object, and the data we needed to acquire was the location of the object in real time. Therefore, we picked *find_object_2d* [7], a package that is able to recognize a user-selected object within a 2D frame and send out the coordinates of the object in real time.

The package comes with a GUI. In the screenshot of the GUI below (Figure 4), the raspberry on the Raspberry Pi case is hand-selected and tracked:

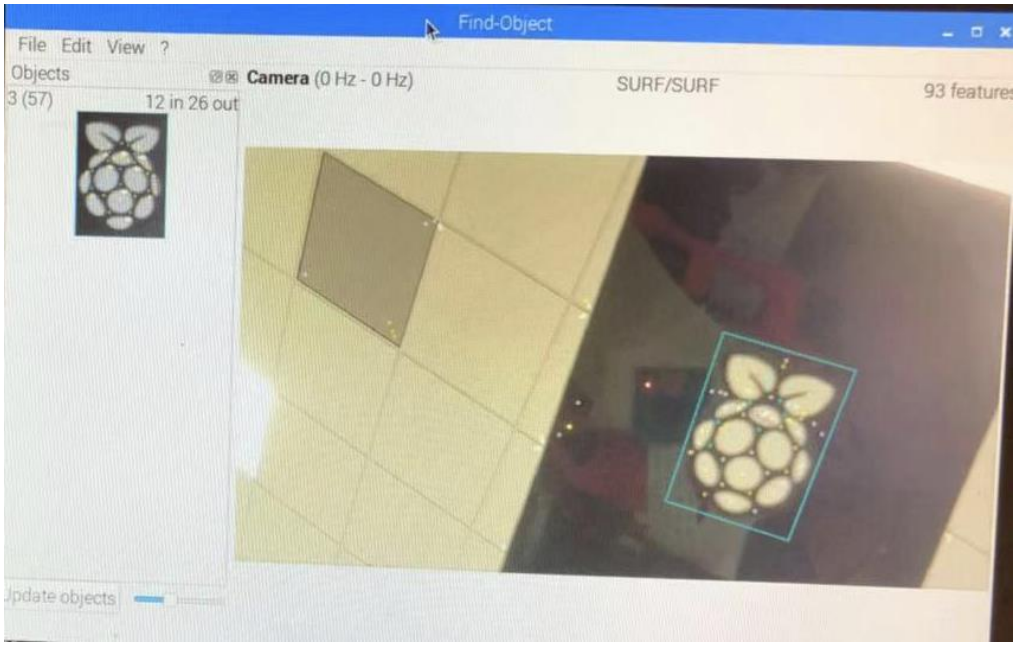


Figure 4: Screenshot of GUI

The *find_object_2d* node publishes on many topics, and the most important one contains the coordinate information of the object being tracked. The message type is *std_msgs/Float32MultiArray* [19]. The elements in the message are based on QTransform [10] and we chose to use the horizontal translation coordinate value for our object tracking algorithm.

2.2. Data Processing

2.2.1. Notes on Writing Subscribers in Python

We noticed that the ROS nodes we had used so far were all written in C++. However, we would like our code to be in Python, to align with the other works already done on the PiCar, and to better reuse the previously developed code on Pi/Arduino communication, which were all written in Python.

ROS supports both Python and C++ equivalently, and the Publisher/Subscriber pairs are not

restricted to the same programming language. Therefore, we decided to rewrite only the Subscriber nodes in Python for both obstacle avoidance and object tracking algorithms, but leave the Publisher nodes unchanged. This required minimum effort and also met what we desired.

One thing to notice is that ROS nodes written in C++ require extra changes in the CMakeList.txt to compile successfully, while Python nodes do not.

2.2.2. Obstacle Avoidance Algorithm

The code for this section is under *Appendix: Obstacle Avoidance*.

Our Obstacle Avoidance algorithm was written in the format of a ROS Subscriber node, subscribing to the *LaserScan* message type, as discussed before.

The Lidar data comes in as an array of [angle-distance] pairs. The algorithm reads in data from the available angles (defined in the YD Lidar launch file), and picks three consecutive angles that are right in front of the Lidar. It determines the existence of an obstacle by comparing the three distances to a pre-set threshold: if all three distances are smaller than the threshold, then an obstacle is detected.

In our algorithm, we set the threshold to be 1.3 meters. It was related to the car speed under normal operation, which will be discussed later. The threshold was picked after several experiments. On the one hand, the car could detect obstacles at a relatively close distance which improves the accuracy of the algorithm. On the other hand, there was still space for the car to safely make its turn without hitting the obstacles.

With the distance data, the algorithm determines whether the car wheels need to make turns. It is not possible to keep sending data from the Pi to the Arduino continuously: the Serial communication channel will get crowded and that will result in huge delay and unpredictable car movements. Therefore, we could only send data when it's absolutely necessary. In order to achieve this, we designed three states for the car wheel turning directions: left, right, and straight. The Pi will only send a signal to the Arduino when the state is changed. Otherwise, the algorithm will stay in the previous state and the car wheels will keep going at the current direction. The car goes straight when nothing is detected, and turns either left or right (opposite to the last turning direction) when some obstacle is detected.

2.2.3. Object Tracking Algorithm

We used the Pi Camera to capture the video stream, and the *raspicam_node* to publish the raw images in the format of ROS image message type (*sensor_msgs/Image*). OpenCV on ROS reads the published message with compatible image type after the *cv_bridge*

conversion, and feeds the message into *find_object_2d* as the input for its functions. If we run *find_object_2d* with GUI, we can see feature points of the objects captured in the frame and a marked region of the specified object being detected.

Our object tracking algorithm uses the message published by *find_object_2d* to enable the car to track any object selected manually. The communication message consists of the coordinates of the detected object relative to the frame, with the message type *std_msgs/Float32MultiArray*. After the object is saved to the detection list, if it is present in the camera frame, the coordinate information will be published. Our object tracking algorithm uses the horizontal coordinate of the detected object presented in the data array.

As discussed before, the Pi-Arduino communication channel could become crowded with too many signals. Therefore, we reused the three-state design in this algorithm. By analyzing the coordinate values received from the publisher, we defined three specific ranges for the object's position in the Camera frame: at the right side, at the middle or at the left side. For each state, the car wheels will turn right, stay straight, or turn left accordingly.

The algorithm *find_object_2d* defines the range for the object's horizontal coordinate to be approximately 0~800, with smaller numbers being the left side and bigger numbers being the right side. There are some fluctuations on the numbers. After several experiments, we defined the coordinate range of the three states: smaller than 280 (left), 280~580 (middle), and bigger than 580 (right). For instance, if the object stays on the right side, with a coordinate value greater than 580, in our code the Pi will send a signal to the Arduino to make the car turn right. Whenever the object moves to other regions, a new signal will be sent to Arduino.

The specific ranges discussed above worked perfect for our car. They could also be adjusted to fit the needs of other different machines.

2.3. Pi-Arduino Communication

The code for this section is under *Appendix: Object Tracking*.

In our project, we used the Serial communication channel for Raspberry Pi and Arduino to communicate, as shown below in Figure 5.

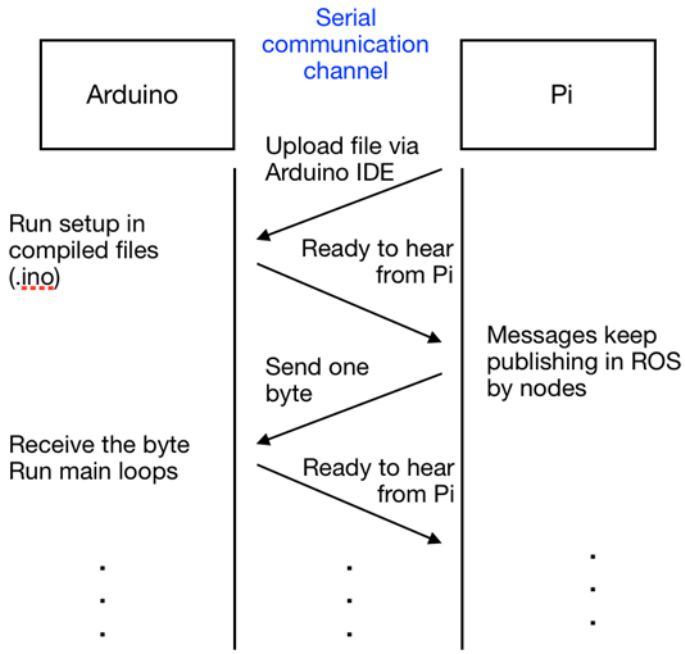


Figure 5: Serial Communication between Pi and Arduino

The communication channel is also indicated in Figure 1 and Figure 7 with the arrows pointing from the two algorithms on the Pi to the Arduino program.

Serial communication is useful for sending or receiving the data from or to the Arduino when interacting with another computer like Raspberry Pi [13]. Before setting up the communication, we needed to check the speed at which the computer communicates with others (written as the following line of code):
Serial.begin(9600)
According to the Arduino documentation, we set the speed for Pi and Arduino to be 9600 [5].

There are a few other important commands we used in the Arduino code [13]:

- *Serial.available()* : gets the number of bytes (characters) available for reading from the serial port.
- *Serial.read()* : reads incoming serial data. It reads one byte (8 bits).
- *Serial.write()* : writes binary data to the serial port. This data is sent as a byte or series of bytes (byte array).
- *Serial.println()*: prints data to the serial port as human-readable text with a line break at the end. We can see the printed messages in the serial monitor.

Take our code as an example,

- *if(Serial.available()){}*

This if statement only runs when there is a serial communication happening between the boards.

- *h = Serial.read()*

The h in this line is a variable of size one byte, which is read via the serial communication channel by the Pi. The Python code on the Pi provides the Arduino with different car movement control signals.

- *Serial.println()*

We also used the serial monitor in Arduino IDE to check the serial communication, and the command *Serial.println()* displays the information the helps us check whether the communication is effective.

A simple example about the Serial communication between Pi and Arduino:
In *camera_object_tracking.py* line 8, we wrote *arduino = serial.Serial('/dev/ttyACM0', 9600),* where */dev/ttyACM0* is the name of the USB interface used, and the number 9600 is the speed of communication between the two board (it should be the same in the Arduino program as well). If the algorithm determines that the object is on the left side, at line 36 in the main loop, *arduino.write(b'1')*, the algorithm on the Pi will send a byte 1 to the Arduino. Then in the Arduino program *CarMovment.ino*, at line 62~66, it takes the byte and assigns the value of variable *h*. This case the car will make a left turn, which corresponds to the Python code line 35~37 in the Pi.

2.4. Car Movement Control

The code for this section is under *Appendix: Car Movement Control*.

2.4.1 Drive Train

We implemented the Motor Feedback Control [20] written by Zou An. The car movement is controlled by the Arduino directly. When the Raspberry Pi sends the order - for example, “turn left” - to the Arduino, the Arduino code will execute the order and control the motor and the wheels.

The setup code should be run once every time before turning on the motor. This part calibrates the reference speed and establishes the control signal for the whole system. The magnitude of the control signal is proportional to the actual car speed, and the signal should be limited to 150~180 (unit: us) so the car could run properly. This is related to the PWM pulse, and the servo motor is at high level when PWM is 150 us. As the last 8 lines in the car movement code shows, the output signal is at HIGH from 1500 (unit: ms) which corresponds to the high level in servo motor, and the signal pulse is created by rotary servo motor. Figure 8 in section 4.1.2 covers the PWM pulse with more details.

In our case, a control signal with 150~180 corresponds to a speed of approximately 1 ~ 1.5 meter/second. Increasing the value of the control signal will bring up the speed of the car.

Without the calibration, the control signal will accumulate when running the main loop in the Arduino program and the speed of the car will increase rapidly. This always happens when the control signal is not "controlled" by any statement and thus takes in the increments during every pulse. We added an if statement (as shown below) to each case to avoid the speeding error mentioned above. (It's actually a hack to the code but we could not figure out another way to fix it.)

```
if(control_signal > 180)
  {
    control_signal = 180;
  }
PPM_output(control_signal);
```

2.4.2 Steering

(The following part refers to the code of car movement in line 55~80)

The main loop in our code only executes when the serial communication is happening. When there is a byte read by the serial port, it sets the byte to be the value of a variable, which is used to determine the turning direction of the wheels. When the Arduino receives an order, the line *myservo.write(angle)* tells the car to turn into that angle. 90 degree is the middle, 125 and 75 are the left and right angles. *myservo.writeMicroseconds()* does the same thing to control the turning angle, with inputs in microseconds. Due to the mechanical limitation of our servo, the turning angle has relatively large uncertainty, which will be further discussed in section 4.1.2.

Note that the control signal represents the reference speed for the actual output signal in the system of motor. The *increment_pid* counts for the error signal increment in the system. We used *delay()* after each wheel turning to control the speed of car.

2.4.3 Power Source

There are three choices to power the board with different pros and cons. The USB port on PC can only support the board working at low power mode, which shows a small lightning symbol at the top right corner on the screen. The mobile power bank can provide enough power and it's convenient for moving tests, but it needs charging frequently. The classical charger is recommended because it can provide enough power for running the board at

normal mode without any issues. However, since the car needs to be connected to an outlet, this method is only for testing the algorithms, without the actual car movement on the floor.

The only power source for the motor is the LiPo battery. The battery should be used with cautions. Overuse will damage the battery permanently because it cannot be recharged once it is used up. The yellow port of the battery should be connected to the motor with correct polarity. The overall schematic of the car, including a mark of the battery, is shown in Figure 6.

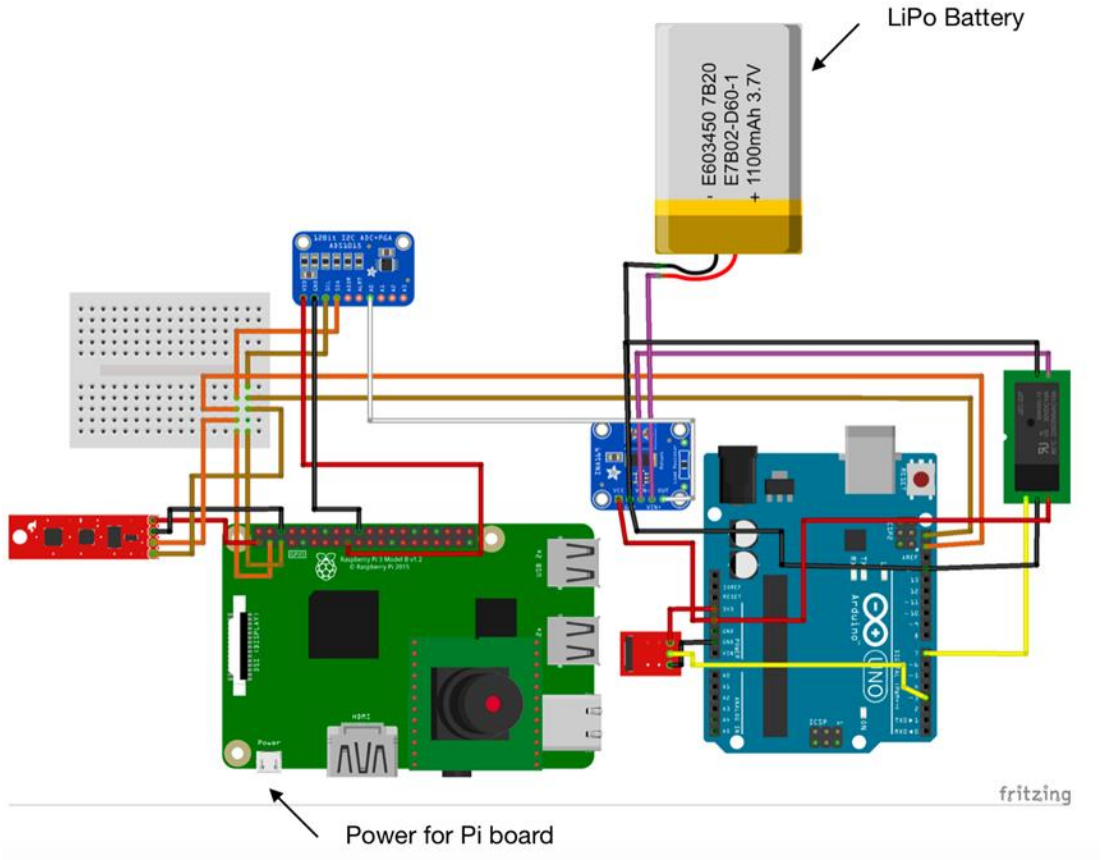


Figure 6: Electrical Schematic with Power Source Marked

3. Results

We developed the following system shown in Figure 7, as an annotated version of Figure 1.

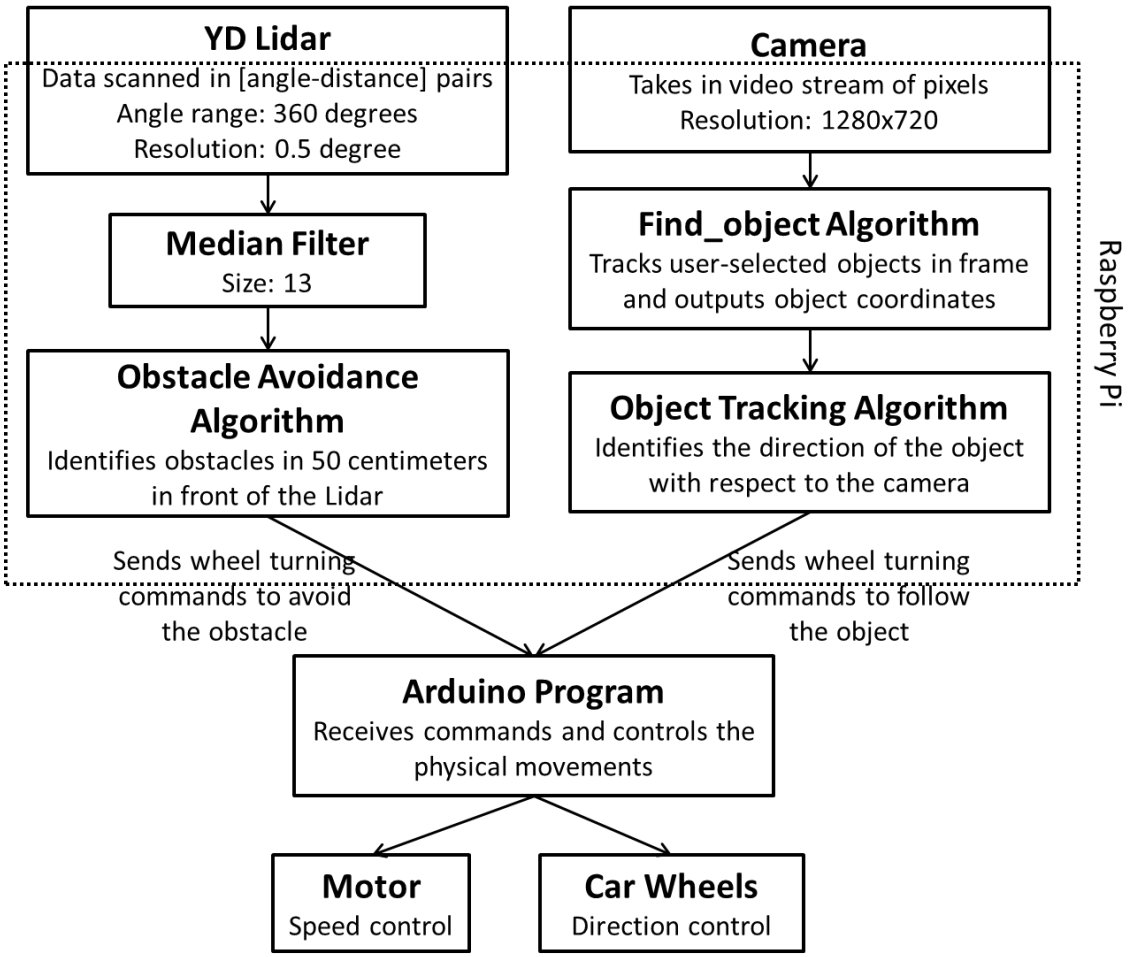


Figure 7: Results

We had separate working versions of obstacle avoidance and object tracking algorithms. The mechanism of the entire system was explained in the previous section.

The car was able to run at a speed of about 1 ~ 1.5 meter/second. With the obstacle avoidance algorithm, it could avoid obstacles of any shape within 1.3 meters, and the obstacles could be either stationary or moving at a speed lower than 1 meter/second. With the object tracking algorithm, it could track a user-selected object of any shape, with a speed similar to the car itself.

The demo videos for both algorithms can be found in the Appendix.

We did not incorporate the two algorithms into one as expected at the start of the project due to the lack of time. Possible future improvements are described in depth in the following section.

Steps for running the programs

1. Set the environmental variables in the terminal, and launch the Publishers to activate the Lidar/Camera.
2. Compile the code on Arduino and wait for the calibration to be done. This will take about 30 seconds. After calibrating for positive, negative and zero RPM, wait a few more seconds before turning on the ESC switch on the car.
3. Make sure the serial monitor is closed before turning on the ESC and running any python programs on the Pi. After turning on the ESC, the car will give one set of regular beeps, which indicates that it's ready to run.
4. Run the Python Subscriber programs. The car will give three sets of regular beeps and start running.

4. Discussion

This section mainly covers limitations, alternatives and future improvements.

4.1. Limitations

4.1.1 Lack of information

Assembly Parts
We found that the assembly list and electrical schematic from the PiCar documentation missed an element on the car. There are two blue rectangular chips on the car, but only one of them (the current monitor) is mentioned in the documentation. We were unable to find any information about the other one.

Usage of Battery
An instruction of using the battery was also missing throughout the documentation but it was essential for running the car. The new battery must be charged before using, and once it goes below 30%, it is permanently damaged and can't be recharged. Therefore, when using the battery, the user must check the remaining charge frequently. The battery might be used up in a couple hours.

4.1.2. Car Wheel Turning Angles Restricted

The car can only turn at certain angles because these angles are stored in the Arduino library of servo. The Arduino library of servo uses PWM to control the angle operations, and PWM signal with a period of 20 ms (frequency 50 Hz) is used to control the motors. Figure 8 below shows the working principles of the PWM-based motor:

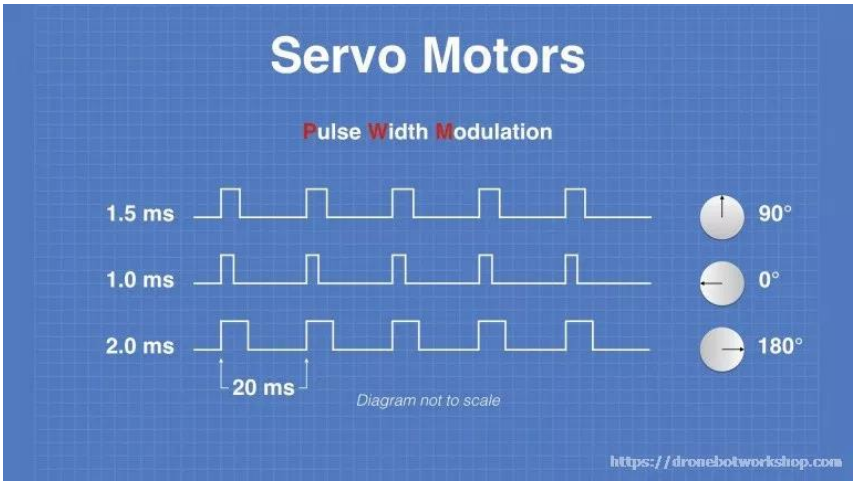


Figure 8: PWM based Motor

- A pulse width of 1.5ms will cause the servo shaft to rest at the 90 degree position, the center of its travel.
- A pulse width of 1ms will cause the servo shaft to rest at the 0 degree position.
- A pulse width of 2ms will cause the servo shaft to rest at the 180 degree position.

Varying the pulse width between 1ms and 2ms will move the servo shaft through the full 180 degrees of its travel. We can modify the pulse width to make the servo shaft stop at any desired angle: if we write *myservo.writeMicroseconds(1500)* (1.5 ms), the servo shaft will set the wheels to mid-point. But the low resolution of the mechanical servo impacts the precision of the turning angles. For example, writing 1.500ms and 1.472ms will result in the same turning angle. [1][2] This leads to the error between the pulse width specified in the code and the actual angle that the wheels turn. Based on our tests, the wheel cannot turn to angles less than 65 degrees or more than 125 degrees, and we can only move the steering in 15 degree increments.

4.2. Alternatives

4.2.1. Using ROS on Ubuntu Instead of Raspbian

As we mentioned before, ROS can also be installed on Ubuntu. We later found that the Ubuntu version of ROS might be easier to use, because it is better developed and maintained compared to the Raspbian version. For example, simpler and clearer commands are used to install ROS packages on Ubuntu, and the process is much more automated compared to Raspbian. We still got our Robot to work on Raspbian, but we might turn to Ubuntu instead next time.

4.2.2. Pi-Arduino Communication Channels other than Serial

We used Serial Communication between the Raspberry Pi and the Arduino board for our project, but there are other alternative approaches: the I2C approach, and the SPI approach.

The I2C was the most commonly used one, and we started off by using I2C communication. At first it worked alright, but after sometime it stopped functioning. From that point on, whenever we tried to send a signal through I2C, it gave the error below:

```
OSError: [Error 121] Remote I/O error.
```

The error kept saying that there was no physical connection between the two boards although we had checked every port on the car. We also tried several approaches to check the connection, including installing the *i2cdetect* tool package and modifying the baud rate on the Raspberry Pi. This error also causes Arduino to skip calibration and speed up the car when other research groups were developing their algorithms based on the existing Car Movement Control code. The issue still wasn't resolved at the end, and that was why we switched to Serial communication.

4.3. Further Developments

4.3.1. Improving Obstacle Avoidance Algorithm

Currently, our obstacle avoidance algorithm is not really sophisticated. It can only tell whether there is an obstacle right in front of the Lidar, but not the ones on its side. Additionally, the next turning direction and the turning angle are also inflexible: only three directions are supported (left, right and straight) and the next turning direction is always the opposite of the previous one.

Possible future improvements for the obstacle avoidance algorithm include:

i) Detecting obstacles in a wider range (maximum 180 degrees);
ii) Calculating the turning direction based on the Lidar input in real time to make turning more efficient;
iii) Improving the granularity of the turning angles.

4.3.2. Shortening Reaction Time of Object Tracking Algorithm

The algorithm *find_object_2d* publishes the coordinates of the object at a low frequency, only once every second or so. Such delay makes the reaction time of our algorithm really long, and limits our goal of tracking object at higher speed and accuracy. This delay still existed after we changed to the lowest Camera resolution and launched the node without the *find_object_2d* GUI. Therefore, we thought it could be due to the pre-set delay within the algorithm, but we couldn't find the exact line to change it in the algorithm yet. If the message of the detected object can be obtained faster, we can improve the tracking feature a lot.

4.3.3. Recognizing Object Rotations and Speed Changes

Currently our algorithm only takes in the horizontal coordinate of the object. However, the information matrix of the detected object uses Affine matrix which has nine coordinates. This Affine matrix comes from the Qt transformation executed by *find_object_2d* on ROS. The message containing the matrix is published under a ROS topic with the message type *std_msgs/Float32MultiArray* (our documentation contains more details about this message type). It helps *find_object_2d* to identify more complicated movements of the object, such as rotations and changes in speed, by providing object shape and size information. This part is done in the algorithm *find_object_2d*, which is useful for our algorithm to detect the changes in the object in higher dimensions.

4.3.4. Combining Obstacle Avoidance and Object Detection

We developed both the obstacle avoidance and the object tracking algorithms, but unfortunately we ran out of time to incorporate them together as planned. As of now the Lidar and the Camera were both working on their own, and there was a lack of communication between the two algorithms. Thus, when the object being tracked is right in the front of the car, the Lidar might label it as an obstacle that the car needs to avoid.

In order to integrate the two algorithms, we need to write another algorithm that handles the communication between the two. The third algorithm should define a coordinate system that works for both the Lidar and the Camera, and incorporate the information gained from the two ends on a single reference frame. It will also handle the communication to the Arduino board. Because the two existing algorithms both output car movement control signals, their priorities need to be defined in the third algorithm as well.

5. Deliverables

i) The Python code of the obstacle avoidance algorithm, in the format of a ROS Subscriber node that subscribes to the *LaserScan* data originally from the Lidar.
ii) The Python code of the object tracking algorithm in the format of a ROS Subscriber node that subscribes to an array of coordinates of the object being tracked, provided by the built-in *find_object_2d* program.
iii) The Arduino code of the car movement control, used to receive signals from the Pi and to control the physical movements of the car.
iv) A video of the car avoiding obstacle in the lab.
v) A video of the car tracking a self-selected moving object in the lab.
vi) A final report incorporating all the progresses taken throughout the project, technical difficulties met, and results.
vii) A final presentation including all the results obtained and the demo videos.
viii) Documentations of the technical approaches used and the algorithms developed, both on the project website and on the PiCar documentation website.

6. References

[1] Deutsch, Stuart. *Arduino's Servo Library: Angles, Microseconds, and "Optional" Command Parameters | Make:* 23 Apr. 2014, makezine.com/2014/04/23/arduinos-servo-library-angles-microseconds-and-optional-command-parameters/.

[2] DroneBot Workshop. *Using Servo Motors with the Arduino*. DroneBot Workshop, dronebotworkshop.com/servo-motors-with-arduino/.

[3] Fernando, Shermal Ruwantha. *OpenCV Tutorial C++*. Color Detection & Object Tracking, June 2017, www.opencv-srf.com/2010/09/object-detection-using-color-seperation.html.

[4] Foote, Tully. *MedianFilter.* ROS Wiki, 26 Jan. 2010, wiki.ros.org/filters/MedianFilter.

[5] Gray, Jeff. *Serial.begin().* Arduino Reference, www.arduino.cc/reference/en/language/functions/communication/serial/begin/.

[6] Gutierrez, Alexander. *Installing ROS Kinetic on the Raspberry Pi.* ROS.org, Sept. 2018, wiki.ros.org/ROSberryPi/Installing%20ROS%20Kinetic%20on%20the%20Raspberry%20Pi.

[7] Labbe, Mathieu. *Find_object_2d.* ROS Wiki, 21 Sept. 2016, wiki.ros.org/find_object_2d.

[8] Lu, David. *Using the Laser Filtering Nodes.* ROS.org, May 2015, wiki.ros.org/laser_filters/Tutorials/Laser%20filtering%20using%20the%20filter%20nodes.

[9] Picar.readthedocs.io. (n.d.). *PiCar Documentation — PiCar Project 2018 documentation.* , picar.readthedocs.io/en/latest/index.html [Accessed 24 Sep. 2018].

[10] Qt Wiki. wiki.qt.io/Main.

[11] Rabaud, Vincent. *Image_view*. ROS Wiki, 18 July 2016, wiki.ros.org/image_view.

[12] RaduBogdanRus. *Cv_bridge*. ROS Wiki, 13 Oct. 2010, wiki.ros.org/cv_bridge.

[13] RAFFIN, Antonin. *Simple and Robust {Computer-Arduino} Serial Communication.* Medium, Medium, 23 Apr. 2018, medium.com/@araffin/simple-and-robust-computer-arduino-serial-communication-f91b95596788.

[14] Robotics. *Ubiquity Robotics Packages: Raspicam_node*. Ubiquity Robotics, 2017, packages.ubiquityrobotics.com/ and github.com/UbiquityRobotics/raspicam_node.

[15] Sebesta, K. and Baillieul, J. (2012). *Animal-Inspired Agile Flight Using Optical Flow Sensing*. Arxiv.org. arxiv.org/abs/1203.2816 [Accessed 24 Sep. 2018].

[16] *Sensor_msgs/CompressedImage Message.* ROS Wiki, Nov. 2018, docs.ros.org/melodic/api/sensor_msgs/html/msg/CompressedImage.html.

[17] *Sensor_msgs/Image Message*. ROS Wiki, Nov. 2018, docs.ros.org/melodic/api/sensor_msgs/html/msg/Image.html.

[18] *Sensor_msgs/LaserScan Message.* ROS Wiki, Nov. 2018, docs.ros.org/melodic/api/sensor_msgs/html/msg/LaserScan.html.

[19] *Std_msgs/Float32MultiArray Message*. ROS Wiki, July 2018, docs.ros.org/api/std_msgs/html/msg/Float32MultiArray.html.

[20] Zou, An. *Motor Feedback Control.* GitHub, 27 June 2018, github.com/xz-group/PiCar/blob/master/src/arduino/motor_feedback_control/motor_feedback_control.ino.

7. Appendix

Demo Videos:

Obstacle Avoidance
https://www.youtube.com/watch?v=uwZC7ohw7J0

Object Tracking
https://www.youtube.com/watch?v=0pLqsSDYCGM

Code for Raspberry Pi:

Obstacle Avoidance
https://github.com/ameliama/PiCar_self/blob/master/lidar_camera_data_process/script/lidar_obstacle_avoidance.py

```
#!/usr/bin/env
python
                import rospy
                import serial
                import time
```

```
from sensor_msgs.msg import LaserScan
#Setup arduino port for the communication
arduino = serial.Serial('/dev/ttyACM0', 9600)
#Define some constants for later use:
#OUTPUT: is there anything in front of the lidar?
NOTHING = 0
SOMETHING = 1
#initialize OUTPUT to be NOTHING
OUTPUT = NOTHING
#LAST_TURN: which direction did it turn last time?
#This is a really simple algorithm that only turns the car to the
direction opposite to the last turning direction when some obstacle is
detected in front of the car.
RIGHT = 0
LEFT = 1
#Initialize LAST_TURN to be RIGHT so it always turns left first
LAST_TURN = RIGHT
#The threshold distance for "seeing" things (how close the obstacle needs
to be for Lidar to detect it)
DISTANCE = 1.3
#Function to convert radius to degeree
def rad2deg(x):
    return (x*180/3.14159)
#The scan function
def callback(scan):
    #Need to define all the global variables
    global NOTHING
    global SOMETHING
    global OUTPUT
    global RIGHT
    global LEFT
    global LAST_TURN
    global DISTANCE
    #Number of sample points taken in from the Lidar. The values are
already filtered by the filter node.
    count = int(scan.scan_time / scan.time_increment)
    #Create an array to store filtered values, initialized to all 0s
    filter = [0]*count

    #For each sample point:
    for i in range (0, count):
        #Take the current value from the LaserScan Median filter
        filter[i] = scan.ranges[i]
        #Calculate degree from radius
        degree = rad2deg(scan.angle_min + scan.angle_increment*i)
        #Again, this is a really simple algorithm and it only examines the
```

```
distance right in front of the Lidar
        if (degree >= 1 and degree < 1.5):
            #If three consecutive distances are smaller than a certain
amount
            #This is used to make sure that something is detected, safer
than only examining one value
            if (filter[i] < DISTANCE and filter [i-1] < DISTANCE and
filter [i-2] < DISTANCE):
                #Write to Arduino and change OUTPUT, only if there is a
change in the state
                #Because Arduino will not work correctly if data is sent
from the Pi too frequently
                #Changing from nothing detected to something detected
                if (OUTPUT == NOTHING):
                    OUTPUT = SOMETHING
                    #Determine which way should it turn
                    if (LAST_TURN == LEFT):
                        LAST_TURN = RIGHT
                        #2 stands for "turn right". Serial communication
only takes in bytes, and we haven't found a clean way to define the int as
a global variable and send it as a byte to the Arduino.
                        arduino.write(b'2')
                    else:
                        LAST_TURN = LEFT
                        #1 stands for "turn left"
                        arduino.write(b'1')
                    print(OUTPUT)
            else: # Changing from something detected to nothing detected
                if (OUTPUT == SOMETHING):
                    OUTPUT = NOTHING
                    #3 stands for going striaght
                  arduino.write(b'3')
                    print(OUTPUT)
#ROS routine for defining a Subscriber Node
def ydlidar_client():
    #Node name
    rospy.init_node('lidar_obstacle_avoidance')
    #Topic that the node subsribes to
    rospy.Subscriber("scan_filtered", LaserScan, callback)
    #Loop
    rospy.spin()
if __name__ == '__main__':
    ydlidar_client()
```

Object Detection
https://github.com/ameliama/PiCar_self/blob/master/lidar_camera_data_process/script/camera_object_tracking.py

```
#!/usr/bin/env
python
                import rospy
                import serial
                import time
                from std_msgs.msg import Float32MultiArray
                arduino = serial.Serial('/dev/ttyACM0', 9600)
                #Define some constants to indicate the location of the object being
                tracked (very rough)
                LEFT = 1
                RIGHT = 2
                STRAIGHT = 3
                #Initialize OUTPUT, the variable holding the state of the turning
                direction, to be straight
                OUTPUT = STRAIGHT
                def callback(array_msg):
                    global LEFT
                    global RIGHT
                    global STRAIGHT
                    global OUTPUT
                    #Getting published data about object coordinates
                    data = array_msg.data
                    #If the object is detected
                    if (data):
                        #Use the 7th coordinate, stored in array index 9, which gives
                information on the object's location on the x axis
                        current_x = data[9]
                        #Determine if the object is on the left, right, or right in front
                of the camera
                        #And update the state of the wheel turning direction to make the
                wheels turn accordingly
                        #OUTPUT will only change when the state is changed. This is to
                prevent sending too many data to the Arduino
                        if (current_x < 300): #Indicating that the object is on the left
                side in the camera image frame
                            if (OUTPUT != LEFT):
                                OUTPUT = LEFT
                                arduino.write(b'1')
                                print "LEFT"
                        elif (300 < current_x and current_x < 600): #If the object is in
                the middle
                            if (OUTPUT != STRAIGHT):
```

```
                        OUTPUT = STRAIGHT
                        arduino.write(b'3')
                        print "STRAIGHT"
                elif (sum > 600):
                    if (OUTPUT != RIGHT): #If the object is on the right side
                        OUTPUT = RIGHT
                        arduino.write(b'2')
                        print "RIGHT"
        def object_track():
            rospy.init_node('camera_object_tracking')
            rospy.Subscriber("objects", Float32MultiArray, callback)
            rospy.spin()
        if __name__ == '__main__':
            object_track()
```

Code for Arduino:

Car Movement Control
https://github.com/ameliama/PiCar_self/blob/master/CarMovement.ino

```
/*
         * Author: AnZou(anzou@wustl.edu)
         * PID speed control for BLCD
         * BLDC(+ESC): TRACKSTAR
         * Encoder: YUMOE6A2
         * Controller: Arduino UNO R3
        */
        /*
         * Pin Definition
         * Pin2 for encoder: interrupt signal
         * Pin13 for BLDC ESC: PPM signal
         *
         *Edited by Chufan Chen and Amelia Ma
         *Fall 2018
        */
        #define reference_signal 35 // set the speed in RPS here
        //set the pid control parameters
        #define K_p 0.1
        #define K_i 0
        #define K_d 0.001
        #include <Servo.h>//servo motor
        Servo myservo;
        int h = 1;//integer for case discussion
        double encoder_counter;
```

```
double motor_speed; //rotations per second
double control_signal;//the output signal from the whole system
double e_k; // variable for P
double e_k1; // variable for I (not in use)
double e_k2; // variable for D (not in use)
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  myservo.attach(9);
  myservo.writeMicroseconds(1500);//set servo to mid-point
  e_k = 0;
  e_k1 = 0;
  e_k2 = 0;
  control_signal = 150; //PWM
  Serial.println("Start!");//start calibration
  pinMode(13, OUTPUT);
  delay(3000);
  Serial.println("Calibrating ESC for positive RPM");
  calibration(500); // Calibrating ESC for +ve RPM
  Serial.println("Calibrating ESC for negative RPM");
  calibration(-500); // Calibrating ESC for -ve RPM
  Serial.println("Calibrating ESC for zero RPM");
  calibration(0); // Calibrating ESC for 0 RPM
}
void loop() {
  // put your main code here, to run repeatedly:
    if(Serial.available()){//if serial communication reads something
    h = Serial.read()-'0';//get the byte from reading and update integer h
    switch(h){
      case 1 ://when byte=1
      myservo.write(115);//turn left
      //delay(100);
      control_signal = control_signal + increment_pid();//calculate new signal
      if(control_signal > 180)//set the control signal to 180 to avoid rapid
speeding
        {
          control_signal = 180;
        }
      PPM_output(control_signal);
      break;

      case 2 :

      myservo.write(75);//turn right
      //delay(100);
```

```
      control_signal = control_signal + increment_pid();
      if(control_signal > 180)
        {
          control_signal = 180;
        }
      PPM_output(control_signal);
      break;

      case 3 :
      myservo.write(90);
      //delay(50);
      control_signal = control_signal + increment_pid();
      if(control_signal > 180)
        {
          control_signal = 180;
        }
      PPM_output(control_signal);
      break;
    }
    }

  //Troubleshooting
  //control_signal = control_signal + increment_pid(); //calculate the control
signal with input signal+error signal
  //PPM_output(control_signal); //output the signal calculated
  //High and low boundary
  //Serial.println("motor speed:");
  //Serial.println(motor_speed);
  //Serial.println("output:");
  //Serial.println(control_signal);

/*if the main loop doesn't work, command the main loop then run
the following loop to move straight*/
    control_signal = control_signal + increment_pid();
    if(control_signal > 180)
      {
        control_signal = 180;
      }
    PPM_output(control_signal); //should be <= 180
    //Serial.println(control_signal);
}
void speed_measure() {
  // measures the motor rpm using quadrature encoder
  encoder_counter = 0;
  attachInterrupt(0, encoder, RISING);
  delay(50); // Wait for 50 ms
```

```
  detachInterrupt(0);
  motor_speed =  encoder_counter * 20 / 200;
}
void encoder()
{
  // increments the encoder count when a rising edge
  // is detected by the interrupts
  encoder_counter = encoder_counter + 1;
}
double increment_pid()
{
  double increment;
  speed_measure();
  e_k2 = e_k1;
  e_k1 = e_k;
  e_k = reference_signal - motor_speed;

  if((e_k>5)||(e_k<-5))
  {
    increment = K_p*e_k;
  }
  else
  {
    increment = 0;
  }

  //increment boundary/saturation
  if(increment > 1)
  {
   increment = 1;
  }

  if(increment < -1)
  {
   increment = -1;
  }

  return increment;
}
void calibration(int command)
{
  int i = 400;
  while(i>1)
  {
    int rate = 1500 + command;
    digitalWrite(13, HIGH);
```

```
    delayMicroseconds(rate);
    digitalWrite(13, LOW);
    delayMicroseconds(10000);
    delayMicroseconds(10000 - rate);
    i = i-1;
  }
}
void PPM_output(int command) {
  int rate = 1500 + command;
  digitalWrite(13, HIGH);
  delayMicroseconds(rate);
  digitalWrite(13, LOW);
  delayMicroseconds(10000);
  delayMicroseconds(10000 - rate);
}
```

Project Webpage:
https://sites.wustl.edu/498picar/