**Autonomous Line Following Car**

**ECEN 4013 Design of Engineering Systems**

**Team 2:**

Jacob Augustyn

Joseph Austin

Roy Baker

Jonathan Ballew

Matheus Barbosa

2 December 2016

**Table of Contents**

1. **Executive Summary**
2. **Design Problem and Objectives**
3. **Detailed Design**
   1. **Assumptions and Justifications**
   2. **System Function**
   3. **Compliance with Engineering Specifications**
   4. **Prototypes**
   5. **Cost Analysis**
   6. **Manufacturing Processes**
   7. **Human Factors**
4. **Laboratory Test Plans and Results**
5. **Bill of Materials**
6. **Gantt Chart**
7. **Results and Conclusions**
8. **References**
9. **Acknowledgments**
10. **Appendices**
11. **Arduino PID Code**
12. **Python Control Code**
13. **Python Line Detection Code - turnDetection.py**

**1. Executive Summary**

This document details the design of a high-speed, autonomous, line-following vehicle that implements 16 infrared sensors for line detection and one camera for incoming turn detection. The vehicle utilizes an Arduino Mega for IR sensor input, steering, and motor control. Additionally, a Raspberry Pi on the vehicle processes image data from the camera module and communicates to the Arduino over a USB connection. For testing and control purposes, the Raspberry Pi also processes input from a Microsoft Xbox 360 wireless controller and sends commands to the Arduino over its serial connection. This allows the vehicle to run IR thresholding setup and start and stop its line following routine remotely.

The track to drive consists of a 25-yard-long rectangular loop of white duct tape with half-meter radius turns. The vehicle is allowed to move off of the line within half a meter on either side. The vehicle described in this report completed the track in 18 seconds on its fastest run.

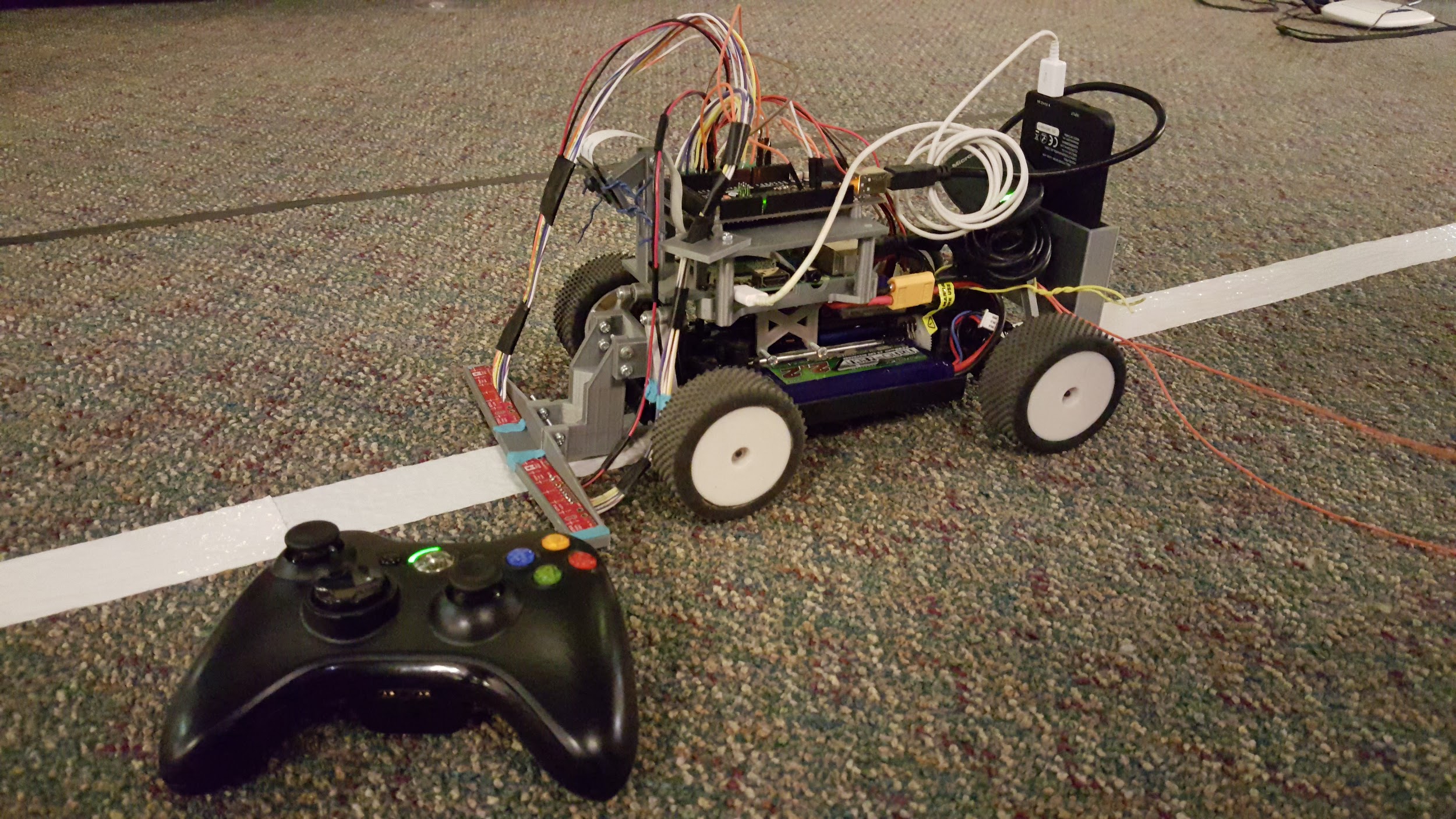


Figure 1: The line-following vehicle and Xbox controller on the track to drive.

This document will describe the design objective to fulfill, the design methodology with justification for part selection and subsystem implementation, the operation of the vehicle, the design process (including prototyping, testing and finalization), and an analysis of the vehicle’s final performance.

**2. Design Problem and Objectives**

The design challenge is to create an autonomous line following car. It is to follow a bright white tape on dark carpet track composed of long straight aways with sharp turns. The track was specified initially to be approximately 10 meters side length square with turns of half-meter radius. The primary design objective is to complete the track as fast as possible, with the fastest vehicle receiving the highest marks for the project. Line following accuracy, however, was not specified.

The design of the car is left up to the individual teams. Teams may design their own hardware using resources available in the ES Design Commons Lab or use pre-built parts and electronic components. Materials may be acquired through the OSU Purchasing Channel with orders given to the course teaching assistant. Each team is given a $330 budget by the university, though small parts costing approximately $10 or less may be purchased by an individual without reimbursement. Upon completion and demonstration of the project, all parts purchased through OSU must be returned to university.

**3. Detailed Design**

**3.1 Assumptions and Justifications**

From the initial design description, the car likely did not need to be able to complete complex paths with arbitrary tun radii, varying degrees of turn sharpness, intersecting paths, changing line widths or color, or discontinuous lines. The track is also stated to be white tape on a dark background carpet. The specification of high contrast implied that the specific color of the line and carpet would not be important, but instead the luminosity and reflectivity of the line and track would prove more pertinent to implementing a line-following solution.

The primary components of the design are as follows:

1. Pololu QTR-8A Reflectance Sensors
2. Arduino Mega 2560
3. Raspberry Pi 3 Model B with Camera module V2
4. 1/16 Scale RC car with 25A ESC, brushless motor, and steering servo
5. AmazonBasics 5000mAh USB battery pack
6. Turnigy 2200mAh 25-50C 7.4V LiPo battery

Reflectance Sensors

The reflectance sensors were chosen for their ability to quickly and accurately read the reflectance of the track. The sensors use IR emitters paired with IR phototransistors to provide an analog voltage readout. High reflectance leads to low voltages at the output, and low reflectance leads to high voltages at the output. As the sensors are designed for line following applications, they were determined to be ideal for achieving the project’s objective. Two sensor array boards were used in order to span wider than the maximum width of the track. Eight sensors per board gives 16 sensors total, providing an expanded range of versus a single board. Wires are soldered to each of the 16 sensors and to the power and ground pins on each board. The 16 sensors are connected to the 16 analog input pins on the Arduino Mega, and the power and ground wires are connected to the battery power and the Arduino ground respectively via a breadboard attached to the car frame.

Arduino Mega

An Arduino Mega was chosen for this design primarily for its native ability to control servos and electronic speed controllers using pulse-width modulation. Moreover, the Arduino Mega has 16 analog ports, allowing 16 analog IR sensor inputs to be processed and factored into the vehicle’s PID control while following the line. Lastly, the Arduino Mega can be powered over the USB connection it shares with the Raspberry Pi, simplifying the power system design on the vehicle.

Raspberry Pi

The Raspberry Pi is well known for being an easy platform to perform image processing with its native camera module support and python support for the powerful OpenCV (Open Computer Vision) library. The use of camera vision was considered as an alternative to the reflectivity sensors for line detection and following but was ultimately decided against as no team member had prior experience with image processing or OpenCV. It was pursued instead as a supplementary system to preemptively detect the turn and notify the Arduino over serial. Early turn signaling allows to car to reach higher speeds on the straight portions of the track and slow down for more accurate turns.

The camera also reads in significantly more visual information than the reflectivity sensors, theoretically allowing it to more accurately follow the line. Should the camera system be developed robustly enough to replace the reflectivity sensors, it would be implemented as such. Though lack of experience and time did not allow the camera system to become developed enough to take on the primary navigation role, it supplemented the vehicle operation with the desired turn-prediction functionality.

Because the The Raspberry Pi has the processing power and bit-width of a 64-bit computer, implementation of a Raspberry Pi in the design allows for simple and effective communication with the Arduino, the creation of a robust testing and operation interface, and data-heavy image processing from an eight megapixel camera. Connecting the Pi to a router and running a VNC session gave access to the Pi anywhere. From the VNC session, the Arduino Mega and camera software are both programmed, and real-time testing feedback is received in the form of images and print statements. A wireless Xbox 360 controller is connected to the Raspberry Pi over USB dongle that allows for manual control of the car during testing, and allows for an immediate motor kill switch should something malfunction or in the case that the car goes too far off track.

USB Battery Pack

The Pi, Arduino, and camera were all powered from a USB battery pack. The battery pack was the Amazonbasics 5v, 5600mAh power bank capable of outputting 2.4A. This was chosen over a BEC because it reduced the demand on the LiPo and was the recommended method of powering the Pi. The 5600mAh capacity provided ample time for testing and came in a small form factor that was easy to mount to the chassis. With the 2.4A output we were able to power both boards and the camera without the need of any additional hardware.

LiPo Battery

The motors, servo, and IR sensors were all powered from a single lithium-polymer battery. A 7.4V, 2200mAh battery from Turnigy was chosen for this role. The two cell LiPo provides 7.4V, enough to provide the 5V as required by the ESC. At 2200mAh, the battery was large enough to provide several hours of runtime turing testing, and the 25-50C discharge rate was sufficient to provide the desired level of acceleration to the motors. The compact size and low weight was ideal for the constraints of the application. The lower price of the battery as compared to larger batteries was also attractive.

RC Chassis

A prebuilt Turnigy 1/16 scale brushless racing buggy was selected as the vehicle for the design. This model came with only the frame, wheels, ESC, drive motor, and steering servo. The vehicle did not come with any electronic hardware for controlling the vehicle. The combination of the high powered brushless motor and the reactive steering servo provided the speed and handling necessary to complete the course and be competitive. Also, the prefabricated chassis cut down on development time and allowed us to allocate more time to software design and testing.

**3.2 System Function**

System Control

The primary motor control of the vehicle is handled through pulse-width modulation voltage signals sent from the Arduino Mega to the vehicle’s electronic speed control and turn servo. Vehicle control functionality is extended to the Pi through the use of commands sent from the Raspberry Pi to the Arduino over the USB connection they share. Commands are integer numbers in which the first two digits indicate the specific command, and four digits following specify an argument to the command. This is done to improve the speed at which commands can be processed on the Arduino, as reading string-type values over serial proved to be prohibitively slow if the devices were to communicate during the line-following routine. Through the use of serial commands, the Pi is able to control the throttle and heading of the vehicle, constants associated with the vehicle’s operation, and signals to indicate the camera has detected a turn approaching.

The following commands and encoding are as follows:

Start listening for commands: 10000

Begin thresholding setup: 10003

Begin line following: 10004

Assert throttle: 12xxx (000 indicates full backward, 500 indicates stop, and 999 indicates full forward)

Assert steering: 11xxx (000 indicates full left, 500 indicates straight, 999 indicates full right)

Increment maximum speed: 13001

Decrement maximum speed: 13000

Stop all operation and stop listening for commands: 19000

Turn incoming: 20000

Turn done: 21000

The Raspberry Pi interprets input from an Xbox 360 controller 100 times per second and sends commands to the Arduino based on the buttons pressed on the controller. While the vehicle isn’t line following, the triggers on the Xbox controller control the throttle, the left thumbstick controls the steering, the bumpers increment or decrement the maximum speeds forward and backward, the back button runs the setup and thresholding routine, and the start button begins the line following routine. While the vehicle is running the line following routine, the B button stops the vehicle and disables command interpretation, and the Xbox guide button closes the program entirely. The use of the Xbox controller and Raspberry-Pi-to-Arduino serial interface allows testing, setup and control of the vehicle throughout the design process.

PID and Speed Control

One of the main controllers of the car is a software-implemented Proportional-Integral-Derivative (PID) controller. This controller uses three levels of error and a control variable to take a system from its current state to a desired state -- often called the set point. The PID controller is the simplest algorithm available that still accomplishes the task properly, giving us three parameters to control in order to achieve the desired process behavior. In particular, it was desired to set up the PID in such a way that the car would not oscillate too much around the line, but would also allow the car to move at a reasonable speed. The car’s position relative to the line, which was read from a pair of infrared sensor arrays, was used as the error fed into the PID controller.

Each parameter (or coefficient) controls a type of error correction, and these error correction “weights” will vary from application to application. In general, PID tuning is done with some educated guessing based on each parameter role. The proportional parameter is as it sounds, it makes a proportional adjustment based on the current error reading. The integral parameter keeps track of past proportional errors and tries to correct for accumulated error. Lastly, the derivative parameter keeps track of how fast the error is changing, this usually prevents “overshooting” the target.

Turn Prediction

The turn prediction is handled by the Raspberry Pi. The program is written in Python using several available libraries, namely PiCamera and OpenCV. PiCamera gives easy access to the camera module hardware used on the Pi, allowing images to be grabbed at specific framerates and resolutions as well as control over camera exposure and correction settings. OpenCV is an open source computer vision library . It contains hundreds of features of which only a few are used in this project.

The Turn Prediction software is outlined below. The source code can be viewed in Appendix C.

1. Image Capture - Image capture is performed in a separate computational thread to reduce wasted computation time, speeding up turn detection. The camera is polled continuously at 45 fps for 320x240p image that are then put into a data stream that can be read from anytime the detection thread is ready.
2. Grayscale conversion - Necessary for thresholding and simplifies computation. Effectively turns the color image into a luminance map. This is beneficial under the assumption of a high contrast line where color is non-essential.
3. Image Blurring - De-noises the image to reduce thresholding artifacts. The carpet has a lot of bright high frequency noise that can reduce track isolation.
4. Image Thresholding - Makes image purely black and white improving contour detection. This takes the high contrast assumption to its extreme end.
5. Contour Detection - Detects shapes in the image which are stored as point sets from which attributes can be calculated.
6. Determine Largest Contour - Isolates the track contour from all other noise. Based on the framing track in the camera, it is assumed that the line is the largest contour.
7. Adjust Threshold Value - Maintains the track contour to be in a certain size range to ensure consistent turn detection. Compensates for the changes in exposure and relative brightness of the track.
8. Scanline - Finds the center of the track along adjustable horizontal lines across the frame. These scan lines mimic simple line scanners with the added benefit of having as many as there are vertical pixels where the only limit is computation speed. The design uses 4 scanlines, which proved to be a good balance of performance and functionality.
9. Turn detection - The standard deviation of horizontal pixel deltas between the track centers is used to determine “curviness” of track ahead. This is designed to allow the car to wobble across the track under PID control without thinking there is a turn. This is valid under the track specifications of constant diameter 90 degree turns.

**3.3 Compliance with Engineering Specifications**

The vehicle met the given specifications. The track was completed within bounds using fully autonomous control. All of the purchased parts remained within budget.

**3.4 Prototypes**

Early Camera based Turn Prediction

Among the first changes to the camera software was the inclusion of threading. Splitting the camera polling thread off into it’s own thread considerably sped up overall processing speed. Other small performance improvements included changing the blur method from gaussian to kernel, and moving the image drawing into another separate thread. Early turn prediction methods relied on several different ways of analyzing the track contour. Attributes such as contour center of mass, bounding boxes, and thickest vertical and horizontal dimensions were used to determine the track location on screen, whether or not it was turning, and which direction it was turning. These various methods proved incapable or unreliable when it came to determining a turn. A few methods were computationally heavy leading to slow processing framerates.

Hardware Mount Prototypes

Mechanical parts were needed on the car to hold our various components. In our early prototyping we had accomplished this by using foam board and perfboard. We then switched to using modeling software and 3D printing our desired parts. The printed parts were then screwed onto various parts of the chassis and were able to accommodate our electronics for testing.

**3.5 Cost analysis**

The project costs totaled at $225.79. This is well within the budget allowed. However, some parts were used that were not purchased through the university, such as the Xbox controller, WiFi dongle, and 3D printing filament. If these parts had needed to be purchased on the project’s budget, the project would probably have been over budget Some reductions could have been made if the Pololu reflectance arrays had been purchased from the beginning, obviating the cost of the earlier IR sensors that were not used in the final design.

**3.6 Manufacturing Processes**

The manufacturing process consisted of 3D printing parts to add to the existing chassis. TinkerCAD was used to design our parts and generate the stl files. The stl files were then ran into Cura 2.1.3 to perform the slicing and generate the gcode for the printer. The gcode files were uploaded to a group member’s personal 3D printer where it produced our final parts. All of the 3D models and parts can be seen in Appendix D.

**3.7 Human Factors**

As the project is an autonomous racing RC car there are few human factors to consider. The most apparent human factor related to this system is the interface for starting and stopping system functions. A majority of the functions are simply controlled via a wireless Xbox gamepad and clear text prompts instructing the user how to use them. This includes a kill switch should the car threaten to collide with a person, their property, or other obstacles. Another human factor we encountered was the fact that all of our part orders had to go through the University. We were delayed by 3 weeks at one point because we had an order that was never placed.

**4. Laboratory Test Plans and Results**

Camera Testing Methods

The first camera tests were performed on single handmade digital files depicting a black line on a white background. These files were used for proof of concept contour detection and analysis. Later camera testing on continuous capture was performed with idealistic lines made from bright white duct tape on a black presentation board. These boards were used to further develop track detection methods and refine performance. Performance was measured using FPS measuring built into the turn prediction thread. After the detection method was finalized and a performance balance was found, the FPS measuring code was removed. Camera testing on the practice tracks set up in the Noble Research Center involved using the XBOX controller and manual driving controls. A live camera feed was viewed over VNC allowing for track detection and camera setting optimization.

PID Testing Methods

Initially, we wanted to try to write our own service functions to read the sensors and map that reading to an error for the PID. The first approach was to read the analog value from each sensor, and then sum all the values weighted by how far out the sensor was (i.e. edge sensors weigh more than center sensors). This value was normalized to a range from -35 to 35, which is the spectrum at which the car can turn (assuming 0 is driving straight). This value was then added to the angle value at which the servo was centered (about 103o) which is what controlled the car steering. Although this method worked for a while, we were a little limited because of the small correction range (75o) and we had no speed control so the car kept the same speed the whole time. This arrangement also proved to be bad when making sharp turns. In the next iteration, we found a library from Pololu which reads the sensors and returns a value of 0 to 15000, 0 being all the way off the line to the left, and 15000 being all the way off to the right.

When we started using the library, this gave us more of an error range to work the PID, and also it made for much cleaner code that ran faster than before. This allowed the PID to make adjustments more often. With this in mind, we retuned the PID by lowering the proportional constant, and also lowering the integral constant. This left us with almost a purely derivative algorithm. At this point, the car’s driving was more precise so we were able to bump up the speed. Our large derivative constant was dampening the oscillation of the car, while preventing overshoot. The relatively low proportional kept the car from constantly over correcting. From then on, we simply made incremental adjustments to the PID constants.

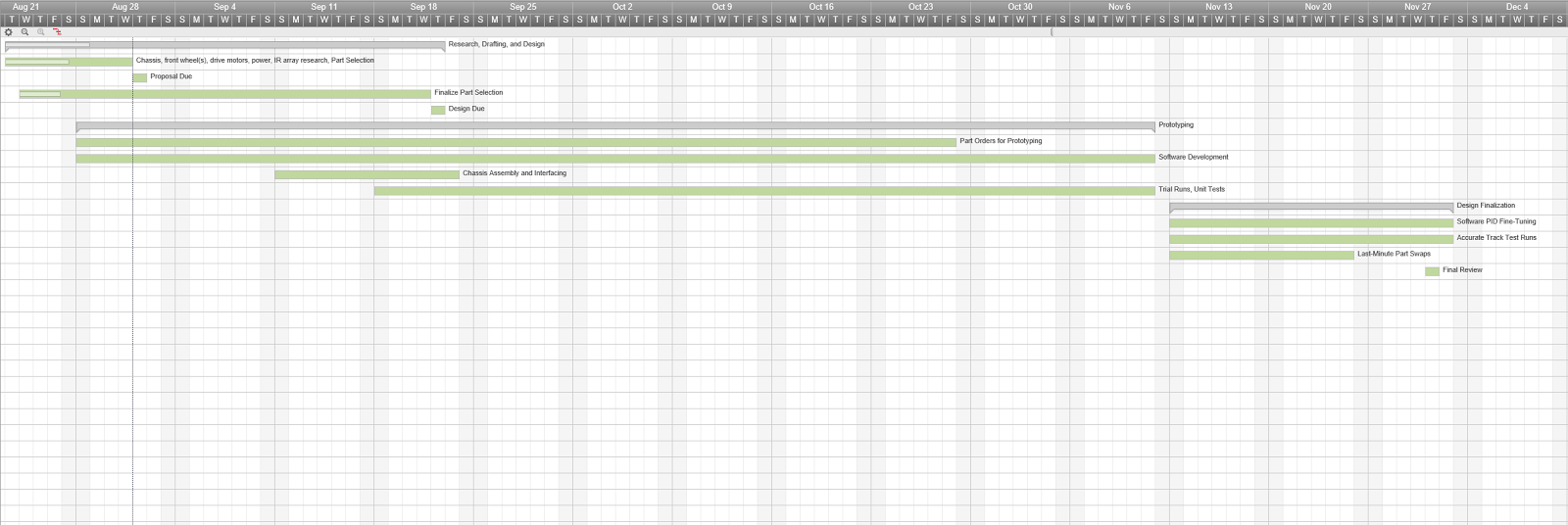
Another problem that we had was with thresholding the infrared sensors. The way the sensors work to detect the line is by thresholding the reflectivity of the color. By using the white line we threshold to determine how much reflectivity is white, but also how much is not white (black). Our trouble was that the thresholding function provided by the library kept setting the maximum reflectivity for white (on the tape) too low. This caused some odd behavior, especially because the carpet on which we had to run had some discrepancies when it came to its color. In order to fix this problem we had to go into the library source code and making some modifications. Instead of completely changing how the library calculated threshold, we instead added a new member function to the QTR sensors class. This new member function took in a factor, and it factored the threshold down by that much. This gave us a new parameter (factor) to manipulate. After incrementally lowering factor, we were able to get the threshold just right, and the car started running well on the carpet.

The final couple of control problems that we had were that the car sometimes would not respond to the PID adjustment quick enough and completely miss the line, also due to the fact that it was moving so fast. Our first fix was to modify the code in order to “pulse” the accelerator instead of constantly writing a speed into the electronic speed controller. Upon further analysis of the code we realized that this was hindering our car’s ability to react because the pulsing was being done by using the delay function, which periodically halts all processes. To fix this, we found a way to make the pulsing without delay. We used the running time of the loop itself as a delay, and kept a free running counter. By the value of this counter we knew how much time had passed, and we could pulse the accelerator without delay. In order to accommodate for the high speeds, we also implemented a condition to “brake” when the error was too high by writing a reverse speed to the electronic speed control.

**5. Bill of Materials**

|  |  |  |
| --- | --- | --- |
| **PART** | **QTY** | **PRICE** |
| Sainsmart Arduino Mega | 1 | $15.99 |
| Raspberry Pi 3 Model B | 1 | $35.99 |
| Raspberry Pi Camera Module V2 | 1 | $26.99 |
| 1/16 Brushless 4WD Racing Buggy w/25A System | 1 | $75.40 |
| Turnigy nano-tech 2200mah 2s 25~50C Lipo Pack | 2 | $10.63 |
| Turnigy b4 Compact 35W 4A Automatic Balance Charger 2~4S Lipoly | 1 | $13.15 |
| HobbyKing 105W 15V/7A Switching DC Power Supply | 1 | $15.05 |
| AmazonBasics Portable Power Bank 5600mAh | 1 | $19.99 |
| Pololu QTR-8A Reflectance Sensor Array | 2 | $9.95 |
| OPB733TR IR Sensor **(UNUSED IN FINAL DESIGN)** | 7 | $2.65 |

**6. Gantt Chart**



**7. Results and Conclusions**

Line Following Accuracy: Moderate. The system stays well within track bounds on straight track areas and comes close to track bounds on turns.

Final Track Testing Times:

Based on unofficial track times, the system is moderately successful. The two fastest teams completed the official track in about 16 seconds while the slowest teams took 25 seconds or more. This system consistently finished laps in about 20 seconds with the fastest lap recorded at about 17 seconds. Line following accuracy is less successful. While no quantitative measurements were taken or specified in the guidelines, the car stayed within the official track bounds in all but the most extreme cases.

The system is overall effective with strengths on the long straight aways and turn prediction. On straight areas where no turn is detected the car has little issue remaining stable, often achieving speeds higher than other teams with similar tracking accuracy. The camera is consistently able to determine when a turn is near and is effective at signalling to the Arduino that it needs to reduce speed. Despite this, the limited range in which the camera is currently able to detect a consistent line is a strong weakness that causes the turn to be detected late. When travelling at higher speeds this increases the chance of turn overshooting and slower recovery of tracking stability.

The largest drawback of the design is its relatively high minimum speed. The chassis and motor system we originally selected was designed for speed, giving poor slow speed control. This often leads to overshooting turns at any speed above our minimum, especially after long straightaways where the system is most effective and easily builds up speed. In the case of overshooting a turn, the system stayed within the track bound a majority of the time.

It is the belief of the team that the system met all design specifications and with more development time the system could reach significantly higher speeds through continued PID tuning and camera software development.

**8. References**

“Arduino Library for the Pololu QTR Reflectance Sensors.” *Pololu.* Pololu.com. Accessed 2 December 2016.

"Arduino Mega Base for Projects by Ijmrtll." *By Ijmrtll - Thingiverse*. N.p., n.d. Web. 02 Dec. 2016.

"Raspberry Pi 3, Pi 2, and Model B Case with VESA Mounts and More by 0110-M-P." *By 0110-M-P - Thingiverse*. N.p., n.d. Web. 02 Dec. 2016.

"QTR-8A (RC) Line Sensor Bracket for Robots by Bitswype." *By Bitswype - Thingiverse*. N.p., n.d. Web. 02 Dec. 2016.

**9. Appendices**

**A. Python System Control**

import os, sys, pygame, serialcomm, cv2, numpy

from turnDetection import PiTurnDetection

from time import sleep

# Display Initialization

pygame.init()

size = 100, 100

grey = 50, 50, 50

screen = pygame.display.set\_mode(size)

screen.fill(grey)

pygame.display.set\_caption('Testing Interface')

pygame.display.iconify()

quitting = False #initiated a window close event

pygame.event.set\_allowed(pygame.QUIT) #only listen for a frame close event

#-----------------------------------------------------------------#

#Interpreting input

pygame.joystick.init()

xbox = pygame.joystick.Joystick(0)

xbox.init()

LEFTSTICK\_X = 0

LEFTSTICK\_Y = 1

RIGHTSTICK\_X = 4

RIGHTSTICK\_Y = 4

RIGHT\_TRIGGER = 5

LEFT\_TRIGGER = 2

A\_BUTTON = 0

B\_BUTTON = 1

X\_BUTTON = 2

Y\_BUTTON = 3

LBUMPER = 4

RBUMPER = 5

BACK = 6

START = 7

GUIDE = 8

DPAD\_HAT = 0;

DPAD\_X = 0;

DPAD\_LEFT = -1;

DPAD\_RIGHT = 1;

DPAD\_Y = 1;

DPAD\_UP = 1;

DPAD\_DOWN = -1;

# Current Values

lsx\_val = 0

lsy\_val = 0

rsx\_val = 0

rsy\_val = 0

rt\_val = -1

lt\_val = -1

DEADZONE = 0.2

VALUE\_MIDDLE = 500

throttle = VALUE\_MIDDLE #1 to 99

steerValue = VALUE\_MIDDLE #1 to 99

#-----------------------------------------------------------------#

#Serial Communication

#Begin INIT for line following with IR

SEND\_START = 10000 #Send this to get the arduino to start listening

READ\_ON = 10001

READ\_OFF = 10002

SET\_THRESHOLD = 10003

SEND\_BEGIN\_FOLLOWING = 10004

SEND\_STEERING = 11000 #last 3 digits = value to set steering to

SEND\_THROTTLE = 12000 #last 3 digits = value to set throttle to

INC\_MAX\_SPEED = 13001 #00 = decrement, 01 = increment

DEC\_MAX\_SPEED = 13000

SET\_P = 14000 #00 = decrement, 01 = increment

SET\_I = 15000 #00 = decrement, 01 = increment

SET\_D = 16000 #00 = decrement, 01 = increment

STOP\_ALL = 19000

TURN\_INCOMING = 20000

TURN\_DONE = 21000

#-----------------------------------------------------------------#

turnDetector = PiTurnDetection().start()

turnDetector.useConst = False

turn = False

comms = serialcomm.SerialTalker()

print("")

print("Press both triggers and release to begin.")

while(True):

event = pygame.event.poll()

if((xbox.get\_axis(RIGHT\_TRIGGER) == -1.0) and (xbox.get\_axis(LEFT\_TRIGGER) == -1.0)): break

sleep(0.02)

print("")

print("Start - Start communication with the vehicle.")

print("B - Stop communinication with the vehicle.")

print("LB - Lower max speed, RB - Raise max speed.")

print("Guide - Stop communication and exit.")

print("Back - Hold to begin IR threshold setup.")

print("Hold Start - Begin line following without setup.")

print("")

def irSetup():

comms.send(SET\_THRESHOLD)

## print("")

## print("Place the vehicle sensors along the track and press A.")

## sleep(2)

## while True: # Poll for an event to read values from the controller

## event = pygame.event.poll()

## sleep(0.2) #5 FPS max, limit debounce

## if xbox.get\_button(A\_BUTTON):

## comms.send(READ\_ON)

## break;

## #if xbox.get\_button(BACK):

## #print("Canceled.")

## #sleep(1)

## #print("")

## #return;

comms.send(READ\_ON)

sleep(0.6)

print("")

print("Place the vehicle along the track and press A.")

while True: # Poll for an event to read values from the controller

event = pygame.event.poll()

## cv2.imshow("im", turnDetector.image)

## cv2.waitKey(1)

#sleep(0.2) #5 FPS max, limit debounce

if xbox.get\_button(A\_BUTTON):

#turnDetector.frame.camera.exposure\_mode = 'off'

turnDetector.useConst = True

#comms.send(READ\_OFF)

#Do Jonathan's thresholding

break;

sleep(0.6)

print("")

print("Place the vehicle centered on the track and press Start to begin line following, or press back to return.")

print("")

while True: # Poll for an event to read values from the controller

event = pygame.event.poll()

sleep(0.2) #5 FPS max, limit debounce

if xbox.get\_button(START):

comms.send(SEND\_BEGIN\_FOLLOWING)

print("Beginning line following.")

break;

if xbox.get\_button(BACK):

print("Values saved.")

break;

print("")

back\_hold\_counter = 0 #hold the back button down for 1 second to start setup

start\_hold\_counter = 0

listening = False #is the arduino listening to the pi?

lineFollowing = False

while True: # Poll for an event to read values from the controller

#cv2.imshow("Image", numpy.hstack((cv2.cvtColor(turnDetector.image, cv2.COLOR\_GRAY2BGR), turnDetector.rawim)))

#cv2.waitKey(1)

event = pygame.event.poll()

sleep(0.01) #100 FPS max

#Turn incoming from camera?

if(turnDetector.turn == True and lineFollowing):

#print("Turn incoming.")

if(turn == False):

turn = True

comms.send(TURN\_INCOMING)

print("Turn incoming.")

if(turnDetector.turn == False and lineFollowing):

if(turn == True):

turn = False

comms.send(TURN\_DONE)

print("Turn done.")

#Tell arduino to start listening

if xbox.get\_button(START) and comms.connected:

if not listening: #only send start command if the arduino isn't already listening

comms.send(SEND\_START)

listening = True

print(">Arduino is listening.")

elif not lineFollowing: #Hold start to begin line following

start\_hold\_counter += 1

if start\_hold\_counter > 100:

print(">Begin line following.")

comms.send(SEND\_BEGIN\_FOLLOWING)

lineFollowing = True

else: start\_hold\_counter = 0

#Hold the back button for two seconds to begin IR setup

if xbox.get\_button(BACK) and comms.connected and listening:

back\_hold\_counter += 1

if back\_hold\_counter > 100:

print(">Begin IR threshold setup.")

irSetup()

else: back\_hold\_counter = 0

#Steering value input

if abs(xbox.get\_axis(LEFTSTICK\_X)) > DEADZONE:

lsx\_val = xbox.get\_axis(LEFTSTICK\_X)

else : lsx\_val = 0

steerValueNew = VALUE\_MIDDLE - int(lsx\_val\*(VALUE\_MIDDLE-1)) #1 to 99, 50 = straight, 1 = right, 99 = left

#Throttle value input

rt\_val = (xbox.get\_axis(RIGHT\_TRIGGER) + 1.0)/2 #Read values as 0 to 1 instead of -1 to 1

lt\_val = (xbox.get\_axis(LEFT\_TRIGGER) + 1.0)/2

throttleNew = int((VALUE\_MIDDLE)+(VALUE\_MIDDLE-1)\*(rt\_val-lt\_val)) #1 to 99, stop = 50

#Emergency Stop

if xbox.get\_button(B\_BUTTON) and comms.connected:

if listening: #only send the stop command if the arduino is listening right now

comms.send(STOP\_ALL)

listening = False;

print(">Arduino is no longer listening.")

if lineFollowing:

lineFollowing = False;

print(">Arduino is no longer line following.")

#Application quit

if xbox.get\_button(GUIDE):

quitting = True

listening = False;

linefollowing = False;

#Increment max speed

if xbox.get\_button(RBUMPER) and comms.connected:

comms.send(INC\_MAX\_SPEED)

#Decrement max speed

if xbox.get\_button(LBUMPER) and comms.connected:

comms.send(DEC\_MAX\_SPEED)

#Update throttle only if it changes

if (not (throttle == throttleNew)):

throttle = throttleNew

if comms.connected and listening:

comms.send(SEND\_THROTTLE + throttle)

#print(SEND\_THROTTLE + throttle)

#Update steering only if it changes

if (not (steerValue == steerValueNew)):

steerValue = steerValueNew

if comms.connected and listening:

comms.send(SEND\_STEERING + steerValue)

#print(SEND\_STEERING + steerValue)

if (xbox.get\_hat(DPAD\_HAT)[DPAD\_X] == DPAD\_LEFT):

print("Dec thresh")

elif (xbox.get\_hat(DPAD\_HAT)[DPAD\_X] == DPAD\_RIGHT):

print("Inc thresh")

if (xbox.get\_hat(DPAD\_HAT)[DPAD\_Y] == DPAD\_UP):

print("Inc")

elif (xbox.get\_hat(DPAD\_HAT)[DPAD\_Y] == DPAD\_DOWN):

print("Dec")

if quitting:

print(">Quitting.")

if comms.connected:

comms.send(STOP\_ALL)

pygame.display.quit()

break

**B. Python Turn Prediction - turnDetection.py**

from picamera.array import PiRGBArray

from picamera import PiCamera

from threading import Thread

import imutils, time, cv2

import numpy as np

class PiVideoStream:

def \_\_init\_\_(self, resolution=(320, 240), framerate=45):

# initialize the camera and stream

self.camera = PiCamera()

self.camera.resolution = resolution

self.camera.rotation = 180

self.camera.framerate = framerate

self.rawCapture = PiRGBArray(self.camera, size=resolution)

self.stream = self.camera.capture\_continuous(self.rawCapture,

format="bgr", use\_video\_port=True)

# initialize the frame and the variable used to indicate

# if the thread should be stopped

self.frame = None

self.stopped = False

time.sleep(1)

def start(self):

# start the thread to read frames from the video stream

Thread(target=self.update, args=()).start()

return self

def update(self):

# keep looping infinitely until the thread is stopped

for f in self.stream:

# grab the frame from the stream and clear the stream in

# preparation for the next frame

self.frame = f.array

self.rawCapture.truncate(0)

# if the thread indicator variable is set, stop the thread

# and resource camera resources

if self.stopped:

self.stream.close()

self.rawCapture.close()

self.camera.close()

Return

def read(self):

# return the frame most recently read

return self.frame

def stop(self):

# indicate that the thread should be stopped

self.stopped = True

class PiTurnDetection:

def \_\_init\_\_(self):

#CONTROL PARAMETERS

self.frame = PiVideoStream().start()

self.useConst = False

self.contours = None

self.lineContour = None

self.image = None

self.rawim = None

self.turn = False

self.stopped = False

self.scanlines = [30, 70, 110, 150]

self.centers = [0,0,0,0]

self.lineDots = np.zeros(320)

self.turnThreshold = 20

self.sleepTime = 0

self.dev = 0

self.avg = 40

self.HIST\_OFFSET = 30

self.CONST\_HIST\_OFFSET = 120

self.expose()

def start(self):

Thread(target=self.detect).start()

return self

def expose(self):

#self.frame.camera.meter\_mode = 'auto'

self.frame.camera.image\_denoise = False

self.frame.camera.drc\_strength = 'off'

self.frame.camera.exposure\_mode = 'auto'

self.frame.camera.video\_stabilization = True

time.sleep(1)

self.frame.camera.exposure\_mode = 'off'

return self

def detect(self):

self.rawim = self.frame.read()

tempImage = cv2.cvtColor(self.rawim, cv2.COLOR\_BGR2GRAY)

self.avg = cv2.mean(tempImage)[0]

while self.stopped == False:

self.rawim = self.frame.read()

tempImage = cv2.cvtColor(self.rawim, cv2.COLOR\_BGR2GRAY)

self.updateAvg(cv2.mean(tempImage)[0])

if self.BadFrame == True:

time.sleep(self.sleepTime)

continue

else:

tempImage = cv2.blur(tempImage, (17, 17))

tempImage = cv2.threshold(tempImage, self.HIST\_OFFSET, 255, cv2.THRESH\_BINARY)[1]

tempContours = cv2.findContours(tempImage.copy(), cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

tempContours = tempContours[0] if imutils.is\_cv2() else tempContours[1]

if len(tempContours) != 0:

lineContourSize = 0

for contour in tempContours:

tempArea = cv2.contourArea(contour, False)

if tempArea > lineContourSize:

self.lineContour = contour

lineContourSize = tempArea

#print(lineContourSize)

if self.useConst == False:

if lineContourSize <= 13000:

self.HIST\_OFFSET -= 2

elif lineContourSize >= 17000:

self.HIST\_OFFSET += 2

bC = cv2.boundingRect(self.lineContour)

for ind, line in enumerate(self.scanlines):

self.lineDots = np.zeros(320)

for x in range(bC[0], bC[0]+bC[2],5):

if cv2.pointPolygonTest(self.lineContour, (x, line), measureDist=False) == 1:

self.lineDots[x] = 1

else:

self.lineDots[x] = 0

center = self.meanIndex(self.lineDots)

self.centers[ind] = center

self.dev = self.turnPred() #np.std(self.centers)

if self.dev == 0:

self.turn = True

elif self.dev >= self.turnThreshold:

self.turn = True

else:

self.turn = False

self.image = tempImage

self.frame.stop()

def turnPred(self):

dif = [abs(self.centers[1]-self.centers[0]),abs(self.centers[2]-self.centers[1]),abs(self.centers[3]-self.centers[2])]

#print(np.std(dif))

return np.std(dif)

def meanIndex(self, array):

weight, mean, total = 0, 0, 0

for index, value in enumerate(array):

weight += index\*value

total += value

if weight/total >= 0:

return int(weight/total)

else: return 0

def updateAvg(self,mean):

if abs(self.avg-mean) > 10:

self.BadFrame = True

self.avg = ((4\*self.avg)+mean)/5

else:

self.BadFrame = False

self.avg = (self.avg+mean)/2

return self

def stop(self):

self.stopped = True

self.frame.stop()

**C. Arduino PID**

#include <QTRSensors.h>

#include <Servo.h>

//----------------------- Vehicle Operation Constants -----------------------//

const int MAX\_SPEED\_FORWARD = 2003;

int currentMaxSpeedForward = 1420; //current max forward speed

const int MAX\_SPEED\_BACKWARD = 699;

int currentMaxSpeedBackward = 1282; //current max reverse speed

const int SPEED\_STOP = 1350;

const int MIN\_SPEED\_FORWARD = 1410;

const int MIN\_SPEED\_BACKWARD = 1300;

const int MAX\_STEER\_LEFT = 133;

const int MAX\_STEER\_RIGHT = 75;

const int STEER\_STRAIGHT = (MAX\_STEER\_LEFT + MAX\_STEER\_RIGHT)/2;

int current\_speed = SPEED\_STOP;

int current\_turn = STEER\_STRAIGHT;

//-------------------------- Vehicle PID Variables ---------------------------//

#define NUM\_SENSORS 16

#define kp 10

#define ki 0.001

#define kd 40

#define MID\_VAL 7500

#define MID\_ANGLE 103

#define SERVO 10

#define ESC 9

#define LEFT 135

#define RIGHT 75

#define LED 7

#define STOP 1350

#define BRAKE 1200

#define GO 1410

#define FAST 1415

#define SLOW 1390

#define NORMALIZE 20

#define FACTOR 25

#define FACTOR\_2 1.0

#define TIMES\_BRAKE 7

QTRSensorsAnalog qtr((unsigned char[]) {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}, NUM\_SENSORS);

float P = 0;

float I = 0;

float D = 0;

int error = 0;

int prev\_error = 0;

int correction = 0;

unsigned int sensors[NUM\_SENSORS];

const int SERVO\_PIN = 10;

const int ESC\_PIN = 9;

int counter = 0;

int HIT\_BRAKE = 0;

Servo turn;

Servo esc;

int offLine = 0; //gets incremented as sensors arrays come

//up empty (off the line), if it gets to

//a value of 2 that means we are completely off

int stopState = 0; //1 for a pending stop, 2 during a stop and 0 for moving

int timeStopped = 0; //time stamp for when we turn off the robot

const int MOTOR\_OFF = 1;

const int WAITING\_TO\_START = 2;

const int MOTOR\_ON = 0;

const int COAST\_DELAY = 500;

//---------------------- Serial Communication Commands ----------------------//

const int NUM\_DIGITS = 5; //Use 5 digit commands

const int VALUE\_MIDDLE = 500; //Percentage inputs from Pi range from 1 to 499.

const int ARG\_INPUT\_RANGE = (2\*VALUE\_MIDDLE) - 1;

const int START = 10; //Start listening to the pi

const int READ\_ON = 1; //Read the middle IR value

const int READ\_OFF = 2; //Read the left IR value

const int SET\_THRESHOLD = 3; //Read the right IR value

const int BEGIN\_FOLLOWING = 4; //Begin line following routine

const int SEND\_STEERING = 11; //last 3 digits = percentage steering

const int SEND\_THROTTLE = 12; //last 3 digits = percentage throttle

const int SET\_MAX\_SPEED = 13; //Increment/decrement max and min speeds

const int SET\_P = 14; //Set proportional constant

const int SET\_I = 15; //Set integral constant

const int SET\_D = 16; //Set derivative constant

const int STOP\_ALL = 19;

const int TURN\_INCOMING = 20;

const int TURN\_DONE = 21;

boolean turning = false;

boolean linefollowing = false; //Running the line following routine?

boolean listening = false; //Listening to Pi?

//---------------------- Pulsing Constants for Driving Slower ---------------//

//const unsigned long GO\_DELAY = 100;

//const unsigned long MAX\_STOP\_DELAY = 800;

//unsigned long time = millis();

//unsigned long stopDelay = MAX\_STOP\_DELAY;

//boolean going = false;

//boolean pulsing = false;

int command = 0; //current command being built

int i = NUM\_DIGITS - 1; //current digits place

//---------------- Setup ---------------//

void setup(){

Serial.begin(57600);

esc.attach(ESC\_PIN);

esc.writeMicroseconds(SPEED\_STOP);

turn.attach(SERVO\_PIN);

turn.write(STEER\_STRAIGHT);

}

void loop()

{

//---- Read In Command and Execute --------//

readCommand();

//------------------------------------------//

//--- Next Interation of Line Following ----//

if(linefollowing) pidControl();

//------------------------------------------//

}

//---- Read In Command and call doCommand ----//

void readCommand()

{

if(Serial.available() > 0) //If there is something in the buffer

{

byte readIn = Serial.read(); //read the ASCII byte

if (readIn <= 57 && readIn >= 48) //If it is an integer

{

int readInt = (int)readIn - 48; //Subtract 48 to start from zero

command += readInt \* pow(10, i); //multiply the number by the tens place we are building at this instant

i--; //decrement the tens place for the next byte we read in

if(i < 0) //if i gets decremented past zero, the command is built

{

doCommand(command); //execute the command

command = 0; //reset command and i

i = NUM\_DIGITS-1;

}//end if(i<0)

}//end if readIn is integer

}//end if(serial)

}

//--------- Compute Command Read In ---------//

int pow(int base, int exponent)

{

if(exponent == 0) return 1; //base^0 = 1

int result = 1;

for (int j = 0; j < exponent; j++)

{

result \*= base; //for as many times as the exponent, multiply the base by itself

}

return result;

}

//---------- Execute Command ---------------//

void doCommand(int commandReadIn)

{

int factor = pow(10, (NUM\_DIGITS-2)); //Distinguish first two digits as command and the remaining as the argument

int command = commandReadIn/(factor); //First two digits indicate command

int argument = commandReadIn%(factor); //Remaining digits indicate argument

float argToValue = 0.0; //interpret the argument as the percentage of max throttle/steering for set throttle and set steering commands

switch (command)

{

case START: //The Arduino is told that it should start listening to the Pi.

switch (argument)

{

case 0:

if(listening) break; //If we're already listening, don't do anything

Serial.println("Started listening.");

listening = true;

break;

case SET\_THRESHOLD:

if ( (!listening)||(linefollowing) ) break;

//Begin threshold setup, turn LED on.

digitalWrite(LED, HIGH);

Serial.println("Begin threshold setup.");

break;

//IR initialization and configuration

case READ\_ON:

if ( (!listening)||(linefollowing) ) break;

calibrateSensors();

digitalWrite(LED, LOW);

//readOn();

break;

case READ\_OFF:

if ( (!listening)||(linefollowing) ) break;

//readOff();

break;

case BEGIN\_FOLLOWING:

if ( (!listening)||(linefollowing) ) break;

Serial.println("Start line following.");

linefollowing = true;

esc.writeMicroseconds(currentMaxSpeedForward); //start going

//Begin Line Following, stop listening to Pi turn and throttle commands,

//but listen for turn detection and all stop

break;

} break; //break case command = START

//----------- Commands to listen for only while not line-following --------------//

case SEND\_STEERING:

if ( (!listening)||(linefollowing) ) break;

argToValue = (MAX\_STEER\_LEFT - MAX\_STEER\_RIGHT)\*((float)argument/(factor)) + MAX\_STEER\_RIGHT;

if(current\_turn != (int)argToValue){

current\_turn = (int)argToValue;

turn.write(current\_turn);

}

break;

case SEND\_THROTTLE:

if ( (!listening)||(linefollowing) ) break;

//Argument from Pi to value to assert by Arduino

argToValue = (currentMaxSpeedForward - currentMaxSpeedBackward) \* ((float)argument/(factor)) + currentMaxSpeedBackward;

if(current\_speed != (int)argToValue)

{

current\_speed = (int)argToValue;

esc.writeMicroseconds(current\_speed);

}

break;

case SET\_MAX\_SPEED:

if (linefollowing) break;

if( (argument == 0)&&(currentMaxSpeedForward > SPEED\_STOP)&&(currentMaxSpeedBackward < SPEED\_STOP) ) //decrement

{

currentMaxSpeedForward--;

currentMaxSpeedBackward++;

Serial.println();

Serial.print("F: "); Serial.println(currentMaxSpeedForward);

Serial.print("B: "); Serial.println(currentMaxSpeedBackward);

}

else if( (argument == 1)&&(currentMaxSpeedForward < MAX\_SPEED\_FORWARD)&&(currentMaxSpeedBackward > MAX\_SPEED\_BACKWARD) ) //increment

{

currentMaxSpeedForward++;

currentMaxSpeedBackward--;

Serial.println();

Serial.print("F: "); Serial.println(currentMaxSpeedForward);

Serial.print("B: "); Serial.println(currentMaxSpeedBackward);

}

break;

case SET\_P:

if(!listening) break;

case SET\_I:

if(!listening) break;

if(argument == 0) //decrement

{

//I -= 1;

//Serial.println();

//Serial.println("Integral: "); Serial.print(I);

}

else if(argument == 1) //increment

{

//I += 1;

//Serial.println();

//Serial.println("Integral: "); Serial.print(I);

}

break;

case SET\_D:

if(!listening) break;

if(argument == 0) //decrement

{

//D -= 1;

//Serial.println();

//Serial.println("D: "); Serial.print(D);

}

else if(argument == 1) //increment

{

//D += 1;

//Serial.println();

//Serial.println("D: "); Serial.print(D);

}

break;

//--------------------------------------------------------------------------------//

//----------- Commands to listen for even while line-following -------------------//

case STOP\_ALL:

if(!listening) break; //If already stopped and not listening, ignore

Serial.println("Stopping.");

esc.writeMicroseconds(SPEED\_STOP);

turn.write(STEER\_STRAIGHT);

P = 0;

I = 0;

D = 0;

listening = false;

linefollowing = false;

case TURN\_INCOMING:

if(!linefollowing) break;

if(turning == false)

{

turning = true;

digitalWrite(LED, HIGH);

//Serial.println("Turning.");

}

break;

case TURN\_DONE:

if(!linefollowing) break;

if(turning == true)

{

turning = false;

digitalWrite(LED, LOW);

//Serial.println("Done turning.");

}

break;

//--------------------------------------------------------------------------------//

default: break;

}

}

void pidControl()

{

int position = qtr.readLine(sensors,1,1);

error = (position - MID\_VAL)/100; //error between 75 and -75. 75 left most sensor on line. -75 right most sensor on line

P = kp \* error;

I += ki \* error;

D = kd \* (error - prev\_error);

prev\_error = error;

correction = (P + I + D)/NORMALIZE;

correction += MID\_ANGLE;

if(correction >= LEFT) turn.write(LEFT);

else if(correction <= RIGHT) turn.write(RIGHT);

else turn.write(correction);

if( (turning) && ( HIT\_BRAKE < TIMES\_BRAKE )) {

esc.writeMicroseconds( BRAKE );

esc.writeMicroseconds( STOP );

esc.writeMicroseconds( BRAKE );

HIT\_BRAKE += 1;

}

else if( (turning) && ( counter % 5 == 0 ) ) {

esc.writeMicroseconds( STOP );

}

else if( turning ) {

esc.writeMicroseconds( GO );

}

else{

esc.writeMicroseconds( FAST );

HIT\_BRAKE = 0;

}

counter += 1;

}

void calibrateSensors()

{

for(int i = 0; i < 100; i++)

{

qtr.calibrate();

}

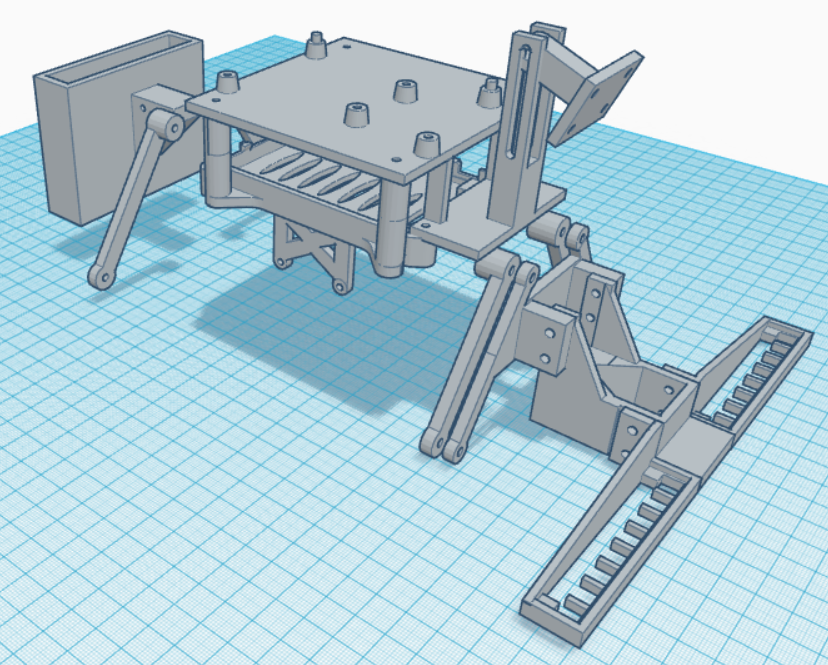
qtr.set\_max( FACTOR );

}

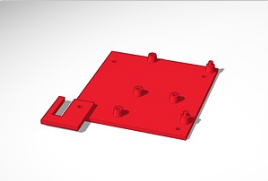
**D. STL Files**

This link will go to Jacob’s TinkerCAD profile where all the files are public domain.

<https://www.tinkercad.com/users/7xnjTiv8obS-jacobaugustyn7>



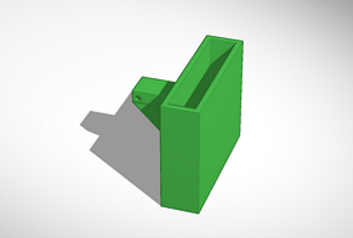
Picture 1: All parts in one model



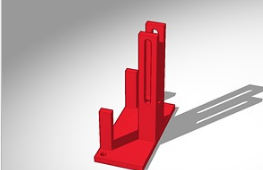
Picture 2: Arduino mount with wire clip



Picture 3: Back strut



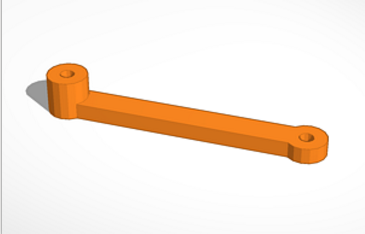
Picture 4: Battery holder



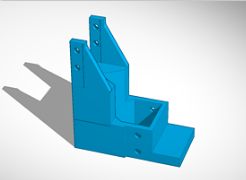
Picture 5: Camera guide



Picture 6: Camera mount



Picture 7: Front strut



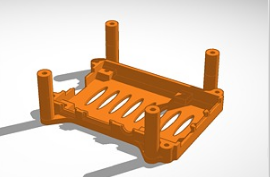
Picture 8: IR arm



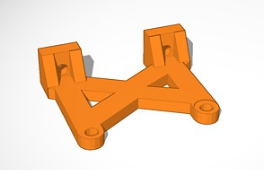
Picture 9: IR array holders



Picture 10: IR base fixture



Picture 11: Pi mount



Picture 12: Standoff for Pi mount