



PyBot, A Remote Controlled Robot built with Python and RaspberryPi

Sylphrena Victori Kleinasser, Lycoming College, Williamsport, PA[†]

This project focuses on the construction of the RaspberryPi remote control robotic unit and creation of an interface that is compatible with machine learning models. The interface has a video feed and will allow control by algorithms or human operators. This project required development of fabrication, practical programming, and micro-controllers skills.

Presented to the faculty of Lycoming College in partial fulfillment
of the requirements for Departmental Honors in Physics

Approved by:

A handwritten signature in black ink, appearing to read 'Chris Kulp'.

Dr. Christopher Kulp, Project Director

A handwritten signature in black ink, appearing to read 'David G. Fisher'.

Dr. David G. Fisher, Honors Committee Member

A handwritten signature in black ink, appearing to read 'Andrew Brandon'.

Dr. Andrew Brandon, Honors Committee Member

A handwritten signature in black ink, appearing to read 'Krish Pillai'.

Dr. Krish Pillai, Honors Committee Member

A handwritten signature in black ink, appearing to read 'Emily Wilson'.

Dr. Emily Wilson, Physics Faculty

[†] ksylphrena@gmail.com
Dated: May 5th, 2023

CONTENTS

I. Introduction	2
II. Methodology	2
II.1. Hardware Choices	2
II.2. Chassis Design	4
II.2.1. Design Availability	6
II.3. Software Design	6
II.3.1. Raspbian Configuration	6
II.3.2. Web Interface	6
II.3.3. Touchscreen GUI	9
II.3.4. Software Availability	10
III. Results	10
III.1. Hardware Complications	10
III.2. Chassis Assembly	10
IV. Future Work	17
Acknowledgments	18
References	18

I. INTRODUCTION

This project is intended to create a well-documented, open-source robot that is easily expandable and modifiable. I built an interface designed to be user-friendly and included features that allow easy implementation of self-driving capabilities. The robot is controlled by a RaspberryPi and uses a camera to stream video to the user. The robot can be used for a variety of purposes, including remote surveillance, remote control, and machine learning research.

Once machine learning models are trained to control PyBot, it would be a useful tool to stream or record in places that are not accessible to humans. Unlike traditional remote control drones and cars, a successful machine learning model could record video even when the device is outside the range of WiFi or cellular networks.

Throughout college campuses and major cities, Starship Technologies and similar companies have recently deployed autonomous food robots. These bots are used to deliver food to customers and are remotely controlled by human operators. This is a potential application of the self-driving capabilities that I will build for the robot in the future, though significant hardware upgrades and further training would be necessary to allow the bot to navigate through city traffic.

II. METHODOLOGY

II.1. Hardware Choices

I spent considerable time on my hardware choices for PyBot. Since the design was incremental and I ordered parts with budgets from my capstone course, Haberberger budget, and departmental funds, I ended up with parts for old designs that I never used in the final revision. In this paper, I will only discuss the parts used in each revision and exclude items that I did not find useful. I discuss complications with parts that I did not use in the final revision in Section III.1, as some of my mistakes may be useful to anyone who recreates my work.

Every part used in the final revisions of each design is listed in Table I, with purchase notes. I opted for more reliable vendors for sensors, motors, and parts, but the budget could be significantly reduced with cheaper sensors and alternate parts.

I chose to use RaspberryPi to control PyBot, as these computers can run standard Linux operating systems and are more versatile than Arduino microcontrollers, which are limited to running code in a specific programming language. Due to supply shortages, the only suitable board available was a RaspberryPi Model 4B, equipped with 2GB of RAM. For this project, more RAM may have been useful, but this model is otherwise great, as it has wide sensor compatibility and good documentation.

There were multiple RaspberryPi camera models available, but given that this car will only be equipped with one camera, I needed a sensor with a wide field of view. I eventually settled on a RaspberryPi Night Vision IR-CUT 5MP camera, which has a 75.7 degree Field of View. The camera board comes with infrared LEDs and the camera does not filter out IR, so the camera has night vision capabilities.¹

Name	Qty	Price	V1	V2	V3
Raspberry Pi 4 Model B (2GB) ^{ab}	1	\$49.14	✓	✓	✓
5" DSI Capacitive Touch Display for Raspberry Pi (800×480)	1	\$43.83			✓
Raspberry Pi Camera Board - Night Vision IR-CUT 5MP	1	\$28.24	✓	✓	✓
120-Piece Ultimate Jumper Bumper Pack (Dupont Wire)	1	\$6.78	✓	✓	✓
Flex Cable for Raspberry Pi Camera or Display - 100mm / 4in ^c	2	\$3.90		✓	✓
ColorFabb Standard Black PLA/PHA Filament - 2.85mm (0.75kg) ^d	1	\$38.95	✓	✓	✓
Adafruit VL53L1X ToF Sensor - 30 to 4000mm ^e	8	\$119.60		✓	✓
Adafruit Ultimate GPS GNSS with USB - 99 Channel 10 Hz ^f	1	\$34.03		✓	✓
Ultrasonic Distance Sensor - 3V or 5V	2	\$7.90		✓	✓
Adafruit DC & Stepper Motor HAT for Raspberry Pi - Mini Kit	1	\$21.57	✓	✓	✓
N20 DC Motor with Magnetic Encoder - 6V with 1:50 Gear Ratio	4	\$50.00		✓	✓
SINGER Dressmaker Pins 00349, Size 17, 500-Count ^g	1	\$30.99		✓	✓
Sandisk 32GB 120MB/s, C10, U1 Micro SD Card with Adapter	1	\$8.35	✓	✓	✓
USB Voltage Step-Up Module 5V to 6-15V	1	\$12.88		✓	✓
STEMMA QT / Qwiic JST SH 4-Pin Cable - 100mm Long	3	\$2.85		✓	✓
STEMMA QT / Qwiic JST SH 4-pin Cable to Female Sockets - 150mm Long	6	\$5.70		✓	✓
STEMMA QT / Qwiic JST SH 4-Pin Cable - 50mm Long	8	\$7.60		✓	✓
M2.5 Hex Spacers, Screws, and Nuts Assortment	1	\$11.99	✓	✓	✓
Two Port 3A 10000mAh Portable USB Battery ^h	1	\$25.99		✓	✓
Anker PowerCore 10000mAh Portable USB Battery ⁱ	1	\$16.99	✓		
USB-A to USB-C Cable 0.4FT 4Pack	1	\$7.99	✓	✓	✓
USB-C to USB-C Right Angle Adapter (3pcs) ^j	1	\$8.99	✓	✓	✓
USB-C to USB-A to USB 3.0 Female to Female Adapter (2pcs)	1	\$9.38			✓
1.0M Flat USB 2.0 Type-A to Right Angle Micro USB	1	\$6.13			✓
USB-A to USB-A Short Extension Cable - 6"	1	\$2.95		✓	✓
Mini Rocker Snap-in Switch - 2 Pin I/O SPST (12pcs) ^j	1	\$6.35			✓
1/4"-20 D-Ring Stainless Steel Mounting Fixing Screw (2pcs) ^j	1	\$6.99			✓
360 Degree Aluminum 1/4" Mount Thread (2pcs) ^j	1	\$5.99			✓
IR LED USB Infrared Floodlight with 1/4" Mount	1	\$28.19			✓
Robot Kit 4WD Robot Car Smart Chassis Kit with 4 Motors	1	\$25.99	✓		
V1: \$224.98, V2: \$454.29, V3: \$561.15					

^a Consider purchasing spare display cables clips from AdaFruit, as the clip breaks easily with frequent removal of the camera/display cable.

^b Note that a model with more memory would be preferable. Without optimizing the operating system memory usage, the Pi uses most of the 2GB memory available.

^c Optional as camera and display include display cables - these are longer and included cables may be too short.

^d One roll is about enough to print all parts once, so multiple rolls may be required to correct misprints.

^e Could switch to ultrasonic sensors to save money, but they are louder and would require design changes. Also, ToF sensors have a narrow field of view, unlike ultrasonic sensors.

^f Could use a cheaper model, but this is a very nice version.

^g Used for 3D printed mecanum wheels.

^h Any battery with two outputs and a decent capacity works as long as it fits - this is not the exact battery I used.

ⁱ V1 uses 5V motors, which don't pull enough power to need two USB ports. The battery bracket is only compatible with this battery.

^j Only one is required, this was the cheapest at time of writing.

TABLE I. List of parts used in all versions of PyBot.

The original design required a very compact battery, but these batteries did not work for the final revision, as I needed two 3A USB ports to power both the RaspberryPi and the motor driver for 6V motors (see Section III.1).

¹ Note that the on-board LEDs are not very powerful, which is why I purchased a supplementary IR floodlight to attach above the camera in my final design. If I did not want night vision, I could have saved money by opting for no floodlight and an IR filtered sensor.

I struggled to pick the Proximity sensors, as there are an overwhelming number of options available with various ranges and specialties. Eventually, I settled on Adafruit VL53L1X ToF Sensors, which have a range of four meters and a narrow field of view. This should simplify simulation for machine learning model training in the future. I also chose ultrasonic sensors.² These will serve to prevent driving off ledges, or brushing the side of the car along a wall. I could have saved money by using entirely ultrasonic sensors as they are a fraction of the cost of ToF sensors, but I was concerned about the wider field of view and noise of those sensors.

Originally, I used a prefabricated chassis which included motors and wheels. This may be a sufficient for a more budget robot, but I abandoned these parts for later revisions—using a prefabricated chassis limited my options for sensor mounting. I switched to N20 DC Motors³, which offer greater flexibility and quality than the bundled items. I used the same motor driver as before, a well-packaged Adafruit DC Stepper Hat, but I switched to 6V motors and needed to use a voltage step-up module (see Section III.1).

Initially, I used [rubber wheels designed for RC cars](#), but they had too much traction to turn without an axle. The wheels worked initially, but with the added weight of all the final components, the motors could not provide enough torque to turn. I switched to use 3D printed mecanum wheels—mecanum wheels are tireless wheels with rollers mounted at 45 degree angles. This allows the robot to move in any direction, and even turn in place. Since I did not have time to design my own mecanum wheels, I used a [remix of a design from Jonah Liu on thingiverse](#). This inexpensive design only requires dressmaker pins for use as axles for the rollers; it is otherwise entirely 3D printed.

Once I switched to 3D printed chassis, I decided to use ColorFabb PLA/PHA as my filament. This is a durable filament, less brittle than normal PLA while maintaining the strength of the material.

II.2. Chassis Design

I made three major versions of the chassis design during this project, each subsequent iteration resolving previous design errors, adding features, and changing styles. More recent designs also illustrate the development of my CAD skills. My early designs used basic extruding methods to hollow out complex surfaces, but using the shell tool creates a much more refined model and wastes less plastic. A comparison of all three chassis designs is shown in Figures 1-3.

For my prefabricated chassis, I designed a RaspberryPi Hat for the camera and battery. I also used this adapter for my second revision, before switching to a closed-chassis design. I planned to create mounts to attach the proximity sensors to the prefabricated chassis, but only printed one mount before deciding that it was unsuitable for the final PyBot. It's also notable that it would have been very difficult to upgrade to the current motors using this chassis, and the provided motor mounts caused drift while driving the car as it only loosely held the motors in place.

I designed PyBot's second chassis from scratch, paying special attention to motor alignment and integrating all sensor mounts into the chassis. I also chose to make this revision highly modular, and to use dovetail joints instead of simple screws to attach sensor mount modules. This complicated the design and printing process. While the chassis worked, it proved to be impractical and was very difficult to assemble with the mounts attached. I could not attach the sensor mounts after attaching the RaspberryPi to the main chassis. Additionally, having the motors stick through the underchassis to the main chassis of the design caused unnecessary stress on the motor circuit boards, though it did fix the loose alignment issues from the first chassis.

For my third iteration, I spent significant time considering how to address flaws from Version 2. For my new design, I lowered the RaspberryPi mount to the underchassis and removed the bottom of the main chassis, lowering the center of mass of PyBot. Portions of the bottom were kept to hold the motors in place and provide mounts for hardware that could not be placed elsewhere. I also scrapped my modular mount design and impractical aversion to screws, melding the mounts into the chassis. To protect the sensors and simplify cable management, I moved all sensor mounts to screw in from the inside of the chassis. Additionally, I improved my design for the front and back sensor mounts, using more advanced CAD methods to properly create a hollow extrusion to allow the ultrasonic distance sensor to point down at an appropriate angle.⁴

In this design, I also switched to a closed chassis, abandoning the battery and camera mount design from Version 1. Instead, I designed a cover with an integrated camera mount, and the battery simply sits inside the chassis. Since I had extra space on top of the chassis, I added a mount for a touchscreen that will be used to display information about the online interface and provide options to connect to new WiFi networks. When I tested previous models, the camera had poor range at night as the IR lights mounted to the camera are not powerful. Thus, I also added a

² Note that I actually purchased cheaper ToF sensors for the side proximity sensors, however, this did not actually save any money. The fact that the short range sensors I selected did not have the capacity to change their I2C communication addresses necessitated my purchase of an [I2C Multiplexer \(I2C mux\)](#), see Section III.1. My robot uses the cheaper sensors and an I2C mux, but I did not list them in my parts list as it would not make sense to reproduce my mistake.

³ Note that these motors have magnetic encoders that can read each motor speed and adjust for manufacturing inconsistencies, but I did not use these capabilities.

⁴ Note that if we did not want depth detection, the design could be simplified by removing the ultrasonic protrusion and lowering the entire chassis to be the exact height required for the proximity sensor mounts, though this might require more careful use of vertical space on the RaspberryPi.



FIG. 1. The original chassis, purchased from an Amazon listing.

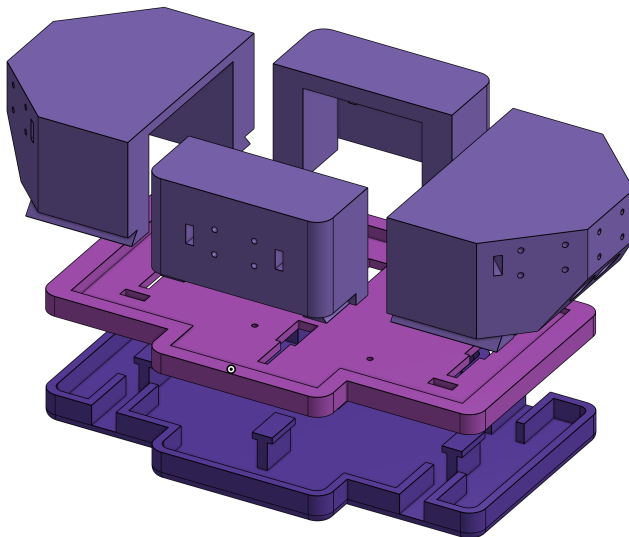


FIG. 2. The second chassis, designed to avoid using screws.

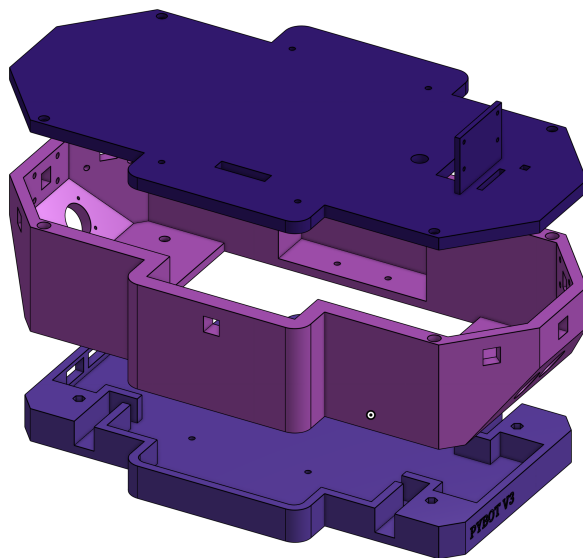


FIG. 3. The current chassis, which greatly improves cable management and space utilization.

mount for an IR floodlight for future use. Finally, I added IO extenders to allow access to usb ports and charging, thus allowing debugging and charging without removal of the top cover.

This design is offers much better cable management, as the main chassis can be easily disconnected from the underchassis even while fully assembled, only requiring a few sensor wires to be disconnected from the RaspberryPi and a few screws. All motor wires can be neatly tucked into the underchassis since I rotated the motors to avoid passing through the main chassis. Most of the sensor wiring can be removed without being disconnected, since the proximity sensors need to pass through the I2C mux before connection to the RaspberryPi. All frequently removed screws attached to metal hex nuts, to avoid striping threaded plastic. The visual appeal of the design is also improved by a closed chassis design and better CAD implementations. While I identify some challenges to the final chassis in Section III.2, none of these issues warrant major redesign or a reprint of Version 3.

II.2.1. Design Availability

The CAD design files used for all versions of this project are available publicly on OnShape at <https://tinyurl.com/pybot-cad>.⁵

II.3. Software Design

II.3.1. Raspbian Configuration

Since this project required many software iterations and updates, I used git to push new code to the RaspberryPi and to handle version control.⁶ I created a Github repository (repo) to host the source code and local copies of the repository were created on the RaspberryPi and on the development computer. With this configuration, code can be edited on the local copy, pushed to the online Github repo, then pulled from the repo to the PyBot. If a bug is introduced with a code change, I can easily see the changes that caused it and roll it back.

Throughout this project, I needed to install many software packages on the RaspberryPi. I used the 64-bit Raspbian Desktop operating system, which is a Debian-based Linux distribution designed for the RaspberryPi. I installed the operating system on a microSD card using the RaspberryPi Imager, and then configured the RaspberryPi to connect to my WiFi network and to enable SSH access.⁷ Ideally, I would not install a full desktop environment as this adds more overhead on the device, but using a desktop environment allows for easy debugging and implementation of the touchscreen interface. In the future, I can optimize the operating system to remove unnecessary packages and services.

Since I ran into several significant roadblocks and switched to different distributions of Raspbian during my project, I found myself reinstalling the operating system on the RaspberryPi several times. Rather than manually configuring the device each time, I automated the process by creating a bash script to install all dependencies, clone the git repository, and configure bash aliases for the Flask application. This script is available in my [Github repository](#).

Additionally, I was using Lycoming College's WiFi network, which does not allow static IP addresses and sometimes blocks SSH traffic. To bypass these network restrictions, I used a VPN to connect to PyBot. Beyond bypassing network restrictions, using a VPN for PyBot would make it possible to connect to mobile networks with PyBot. I used [NordVPN's MeshNet feature](#), as I had a subscription to the service. NordVPN offers this service for free, so no VPN subscription is required to use this feature. Other VPN services may be viable alternates, or one could configure their own VPN.

I created a simple Python script to find the current IP address on a specified WiFi interface of PyBot. The script messages the IP address to a personal discord server and updates the displayed IP address for the touchscreen GUI (see Section II.3.3). Then I created a [systemd](#) service to run a bash script on system boot. This bash script runs the IP Python script, serves the touchscreen interface with Python, and opens the touchscreen interface in Chromium (see Section II.3.3).⁸ The script is included in the [Github repository](#) and the systemd service is added with the setup script described above.

II.3.2. Web Interface

After the operating system was configured, the next step was to create Python functions to control the motors. The Adafruit libraries do not include driving logic, they just provide capacity to set motor throttles from -1.0 to 1.0 for

⁵ Onshape versions are immutable, so I cannot fix issues on old design versions. The Onshape designs are the versions I used to print each revision of PyBot, mistakes and all. Any known issues are fixed in the main branch.

⁶ Git is an industry standard version management system which documents changes to code and keeps records of old versions.

⁷ SSH is used to remotely access the Raspbian terminal, to learn more about SSH visit [Wikipedia](#).

⁸ Systemd is a Linux software suite that can run programs as services when a computer boots.

each of the four motors. To move forward and back, I set all throttles to the same speed, either negative or positive. PyBot V3 can turn extremely sharply, as it used mecanum wheels. For a visualization of how mecanum wheels work, see Figure 4. To turn, we set the throttle of each side in opposite direction, causing the car to turn in place (scenario e). If PyBot needs to turn while moving forward or back, the throttle of the left or right motors is simply set to be 0.0 (scenario d). I did not implement the other movement options (scenarios b, c, and f) to limit complexity of my keybindings.

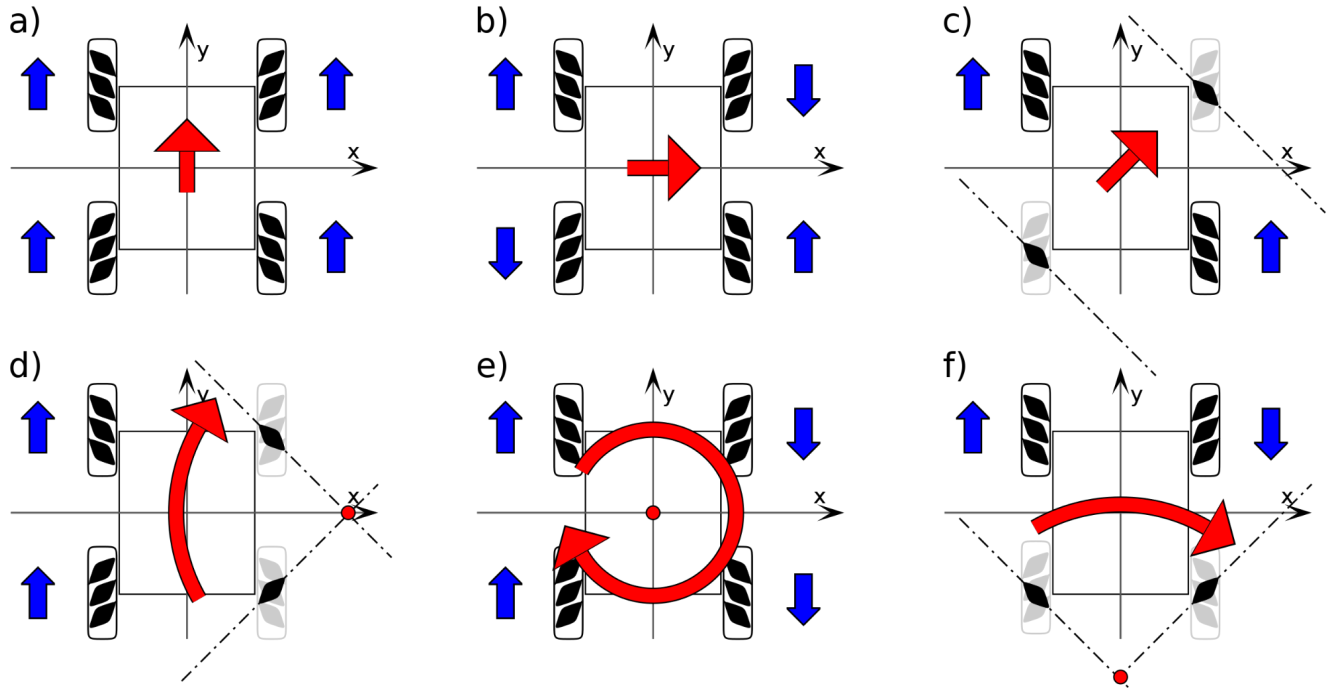


FIG. 4. Mecanum wheels, which allow movement in any direction: a) Moving straight ahead, b) Moving sideways, c) Moving diagonally, d) Moving around a bend, e) Rotation, f) Rotation around the central point of one axle [1].

After the motor functions were developed and tested, I needed to create a web interface to control PyBot and interface with the operator. I decided to use a web framework, Flask, to control PyBot. Flask is a web development framework that implements user configurable HTML, CSS, and JavaScript, controlled by the Python backend. This means that the web interface can have the flexibility of a normal website and maintain compatibility with Python-based machine learning algorithms and Adafruit sensors.

I adapted a previous Flask project for PyBot to take advantage of previous time spent styling the basic interface. I configured an SQLite3 database to implement login authentication and password protection for pages, so that the web interface and controls are not exposed to the potentially hostile users, and to store settings between sessions.

The most challenging part of the Flask configuration was implementing the video stream into the interface. After experimentation with numerous techniques, I decided to use the PiCamera2 library, which has documentation on providing camera streams while allowing still image captures for use with machine learning models [2]. The modified implementation was much faster than the other options I explored with the original PiCamera library. The implementation allows access to the interface from multiple users and might permit simultaneous control of the throttle, but the camera will not cause an exception.

Unfortunately, configuration from this point became even more challenging. For simplicity, I decided to use arrow keys to control the car. This meant that JavaScript had to be used to detect key press events and execute a Python command. Originally, I had JavaScript sending commands to the Python backend e.g. “turn right” or “go forward,” but controlling the motors from the client code invited errors and bugs as the JavaScript code is running on the client’s browser and can easily give conflicting commands. Instead, I chose to send the arrow events to Python and do all handling from the Python backend.

On the client, a `keyDown` event is triggered in JavaScript anytime a key is pressed and a `keyUp` event is triggered when it is released. Unfortunately, `keyDown` events continue to fire every few milliseconds until released, and sending many events in a short interval can cause the Python server to crash. I implemented code to ignore all duplicate `keyDown` events, which almost solved the problem—as I will explain below.

JavaScript sends information to the Python server using HTTP GET requests, which are sent to an endpoint URL

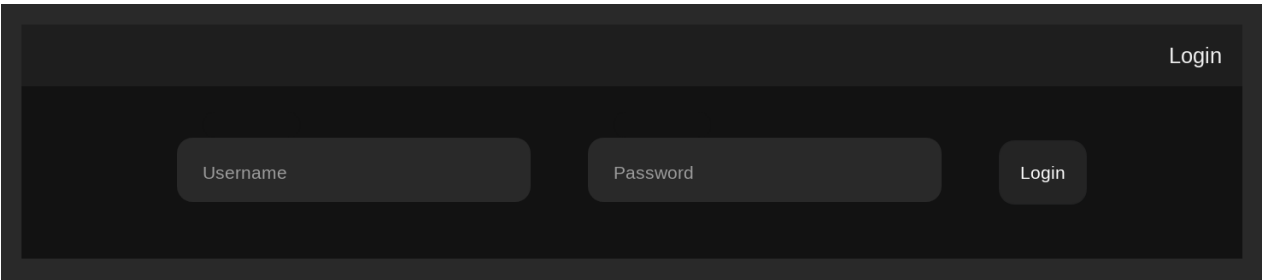


FIG. 5. PyBot's login page.

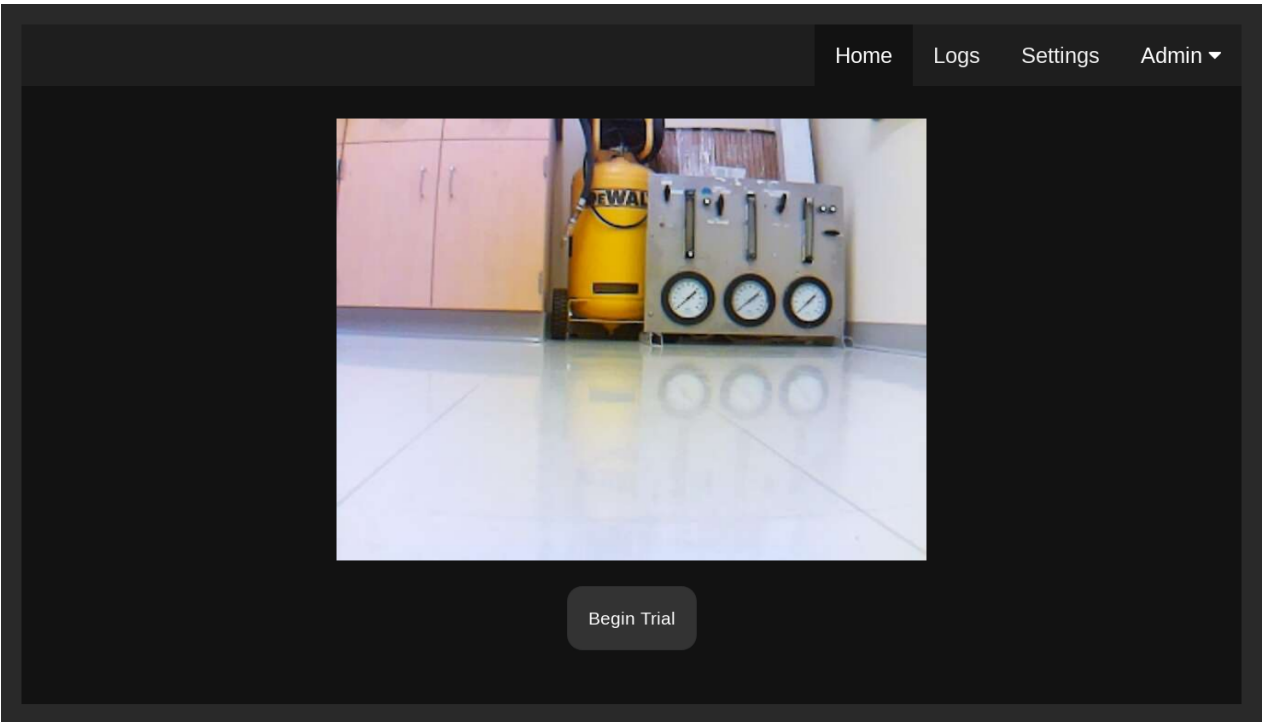


FIG. 6. PyBot's web interface.

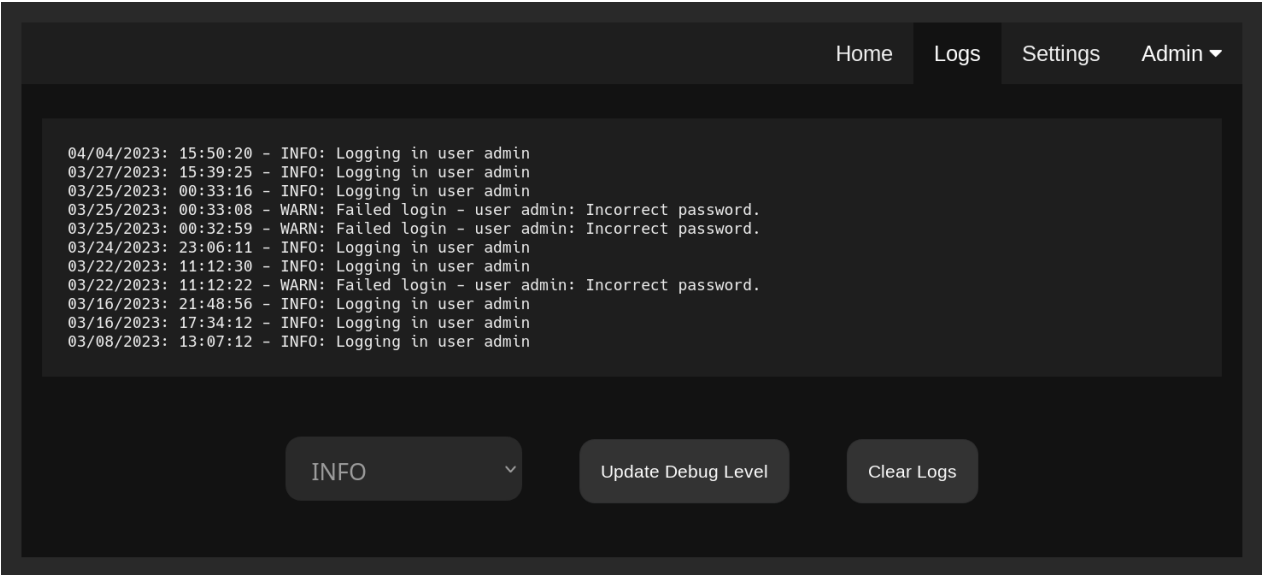


FIG. 7. PyBot's logging page.

on the Python server. The Python code watches this endpoint, and data can be passed to Python via arguments in the HTTP request. As an example, the following GET request might send when the left arrow is pressed down: `/move?arrow=left&state=down`. To prevent unauthorized users from using these GET requests, a user to be logged in for Python to receive these requests.

I configured the code to prevent conflicting commands—if two conflicting keys are pressed simultaneously, the second key press will be ignored. If two keys are being pressed, any further keypresses will be ignored. When Python received a `keyDown` event, it will store the key in a list to maintain a list of active keys, which allows the code to ignore excess keypresses.

I had an ongoing issue during the development of this program, where the motors would get “stuck” running after all keys were released. I discovered that this was an edge case in my code—when combinations of keys are pressed, sometimes JavaScript lets duplicate key events register, even when told to ignore duplicate events. My code assumed that JavaScript would not allow this to happen and would insert a duplicate arrow in the active list, so when the `keyUp` event occurred, a second key would remain in the array, causing the throttle to remain engaged. While hard to diagnose, I quickly fixed this by checking if an arrow already exists in the list before adding it to the list of active keys.

At this point, the car was fully functional and the interface worked great. I added user settings to the interface to allow registration of new users and allowed changing the interface username and password. I made the user registration page require an authenticated user and added a default user with a generic password. I also added a settings menu to configure camera and motor settings. To implement this feature, a new SQL database table had to be added to store these settings between sessions. I also added a page and SQL database table to store PyBot logs.

I also implemented an option to record trials for use in the training machine learning models (see Section IV). This was not too difficult to add to my existing code. First, I added a button to the interface to start/end the recording. When this button is pressed, the backend runs a function that creates a trial directory under the main folder of the application, identified by timestamp recorded when the button was pressed. The function also creates a CSV file to store the keypresses. JavaScript also starts calling the backend, taking a picture every 1000 milliseconds and saving this to the trial directory. This interval will likely need to be increased, but it is a good starting point. When a trial is active, any movement command that is sent to Python is recorded in the CSV file. When the button is pressed again, the backend stops recording and closes the CSV file.

The completed interface is shown in Figures 5-7. I did not include a screenshot of the settings page, as I did not have time to complete the settings page prior to the publication of this paper. The control page is completely functional, but I plan to add a visualization of the proximity sensor data, options to record proximity data in trials, implement the settings on the settings page, and add controls to turn off or restart PyBot.

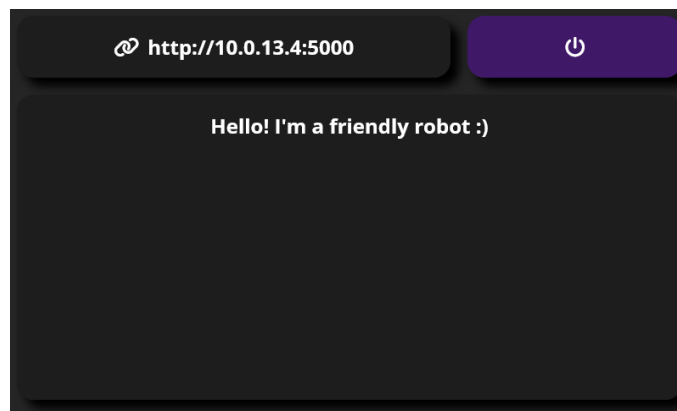


FIG. 8. PyBot’s touchscreen interface.

II.3.3. Touchscreen GUI

I also implemented a simple interface for the screen, using static HTML/CSS/JS. The interface is shown in Figure 8 and displays the current IP address, a message, and a power button. The message and IP addresses are fetched from text files, which are updated by the web interface and Python IP notifier script respectively.

II.3.4. Software Availability

The source code used for this project is available publicly on github at <https://github.com/sylphrena0/pybot>. This repository also includes the source files for this paper, images, and documentation for the code.

III. RESULTS

III.1. Hardware Complications

I ran into several hardware complications during the assembly of PyBot. Rather than unnecessarily extending Section III.2, I will explain the complications in this section. These issues caused delays, as several of them required more parts or work than projected. Anyone that might replicate my work should be take caution to avoid these roadblocks.

One of the first complications in my plans involved the proximity sensors. Digging through documentation and projects, I discovered that using multiple I2C controlled sensors of the same type in a project caused problems. Each proximity sensor on the PyBot is read over I2C, a protocol that allows the RaspberryPi to read data from the sensor. Unfortunately, I2C use addressing, and RaspberryPi does not have multiple I2C channels. The proximity sensors listed in Table I can change their addresses every time the RaspberryPi boots, all of the other sensors must simply be turned off with their a communication pin called ‘XSHUT.’

Unfortunately, when purchasing my parts, I chose to use cheaper sensors for the side proximity sensors. These sensors are shorter range and do not have the capacity to change their address. This forced me to purchase an I2C Multiplexer (I2C mux). My I2C mux can accept up to 8 sensor inputs and send all the sensor data to the RaspberryPi I2C communication pins under a different address. While I use an I2C mux for my design and have a mount included in my CAD files, I recommend that anyone replicating my work to purchase the more expensive proximity sensors that can change their address. This will save time and money.

My next challenge was a more subtle issue. When I first assembled my robot, I configured the motor driver board to pull power directly from the RaspberryPi. However, when using the voltage step-up board to provide 6V to my motors, I noticed that the RaspberryPi would frequently crash. After some deliberation, I surmised that the issue was that the motor driver was pulling too much power from the RaspberryPi, causing the computer to undervolt and crash. This was a simple fix, but it required me to abandon the original battery packs, as they only had one output port. I switched to a battery pack with two 3A USB-A outputs and connected my motor driver to the second port.

Replacing the battery did not resolve all of the power issues. I continued to encounter crashes with the new battery, though slightly less frequently. It seemed that the battery pack would stop providing power when the motors were under load. I tried several different battery packs, but the issue persisted. This was unexpected behavior, as the battery packs were rated for 3A output, and the motors are rated to pull no more than 0.2A 6V at stall torque, which is significantly less power draw than the 3A output of the battery pack. Eventually, I discovered that the voltage step up module is known to pull more than 3A at peak, even when the device it is powering does not pull anywhere close to that current, even though it is advertised to only draw 3A. This meant that the module was drawing more power than the battery pack could handle, so the battery pack would power cycle, crashing the RaspberryPi. I replaced the voltage step up module with a different model, and this resolved the power issues. The new voltage step up module is the one listed in Table I.

My final complication was also related to my motors. The motors are placed in channels in the underchassis. Unfortunately, the clearances are very tight and the motors have a spinning portion on the back. If anything touches the back of the motor, it requires almost no force to prevent the motor from spinning entirely. When the motors are placed in the chassis, the spinning portion of the motor is very close to the channels in the chassis. This means that when the wheels are attached and have weight on them, they can press against the chassis and prevent the motors from spinning. This was a simple fix, I simply added a small spacer between the motor and the chassis to prevent the motor from touching the chassis. This is included with my design files, available in Section II.2.1.

III.2. Chassis Assembly

I won’t mention my second design revision for PyBot in this section as it has no value over the final revision, though I will briefly cover the prefabricated design, since this is a decent option for a budget PyBot, if creature comforts are not a concern.⁹

⁹ If anyone replicates my work with a prefabricated chassis, the turning radius can be improved with mecanum wheels.

Since I iteratively designed the chassis and software concurrently, the assembly process was also iterative. I never attached all the sensors and modules until the final chassis revision, as each time the assembly process revealed design quirks that needed to be fixed before proceeding.

The PyBot V1 assembly was not too complicated:

1. First, I assembled the chassis according to the manufacturer's instructions, soldering leads onto the included motors and attaching them to the chassis.
2. I installed the RaspberryPi on the mounts in the chassis, using hex spacers to provide space for wiring. I also attached the motor driver board,¹⁰ which screws on top of the RaspberryPi and connects to the General Purpose Input/Output or 'GPIO' leads.
3. Next, I connected the motor cables to their respective connectors on the motor driver. Since the motor driver is powered separately from the motors themselves, I connected the power to the RaspberryPi 5V power and ground leads.¹¹
4. I designed and 3D printed a camera and battery bracket, which screws on top of the motor driver. I attached the camera to the bracket with screws, attached the battery with adhesive velcro, and screwed the bracket in place. I attached the camera ribbon cable to the RaspberryPi and added a GPIO cable to control the night vision of the camera.

The completed assembly of the prefabricated design is shown in Figure 9. No additional sensors or accessories were implemented for this chassis.

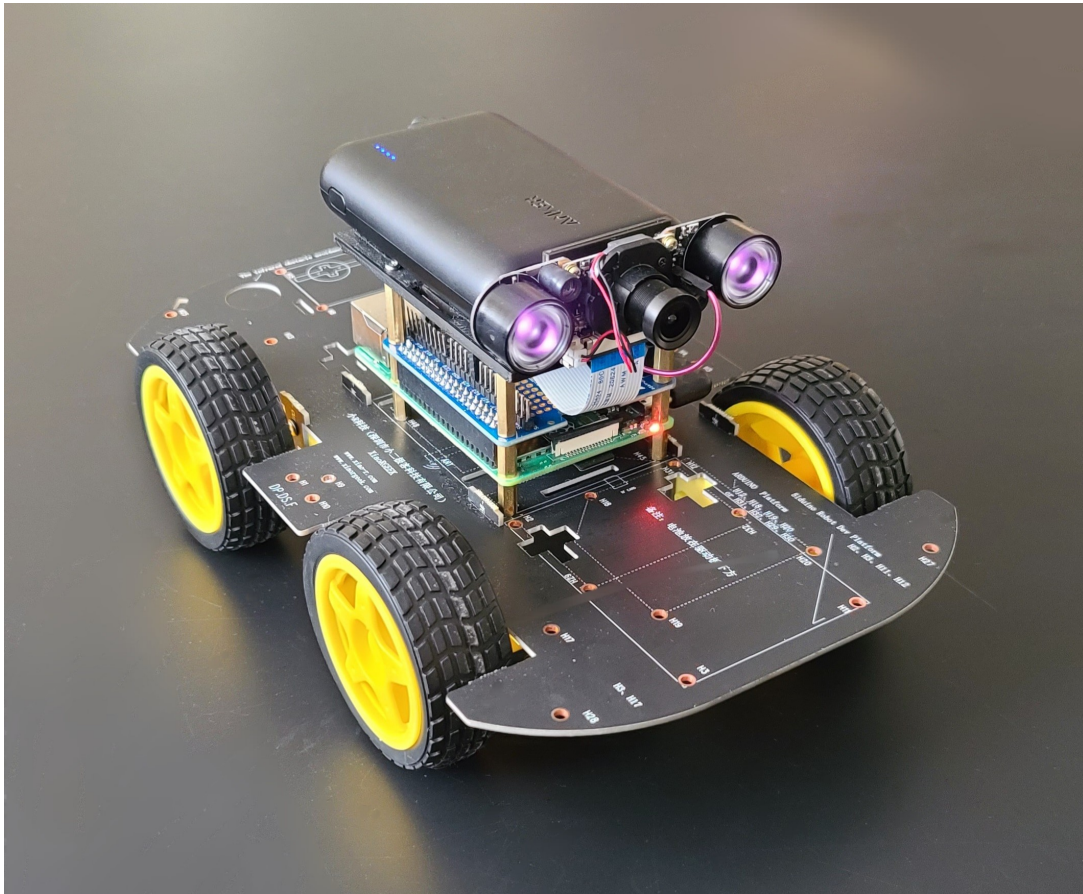


FIG. 9. PyBot V1, assembled on a prefabricated chassis.

For PyBot V3, the assembly is more complicated as there are many more sensors and parts at play:

¹⁰ I had to first solder the GPIO headers and motor/power connectors to the board, this is common for RaspberryPi accessories.

¹¹ As mentioned in Section III.1, this caused problems later. If multiple power sources are available, the motor power should not be routed through the RaspberryPi.

1. First, I printed my designs for the chassis. As shown in Figure 3, the design has three pieces—the underchassis, the main body, and the cover. I also 3D printed and assembled the mecanum wheels according to the instructions on the designer’s [thingiverse page](#).
2. Once the chassis is printed, I inserted hex spacers into the screw holes for the underchassis and main body. The underchassis spacers are screwed in from the bottom of the chassis, while the spacers for the main body use hex spacers with screw ends to screw directly into the plastic of the walls.¹²
3. Next, I placed the IO extensions (USB-C to USB-A female to female adapters) in their holes in the underchassis, holding them in place with heavy duty adhesive, then attaching a USB-A male to USB-C male cable and USB-A male to micro USB male cable to the USB 3.0 and power adapters respectively. I soldered wire leads to one of my rocker switches and set that in place.
4. With the USB extensions in place, I attached short hex spacers to the RaspberryPi attachment holes, then screwed in the RaspberryPi. The spacers allow the IO extensions to pass underneath the RaspberryPi for cable management. I attached the USB 3.0 data IO extension to one of the RaspberryPi USB 3.0 ports, and wound up the excess micro USB cable in the front of PyBot for later use. I also attached the camera ribbon cable to the RaspberryPi, as it is inaccessible once the motor driver is attached.
5. I placed the motors into their alignment slots, then I attached the AdaFruit motor driver to the the RaspberryPi¹³ and attached all the red and white motor power leads to the power and ground connectors of the motor driver.¹⁴ I added the spacers mentioned in Section III.1 to prevent the spinning portion of the motors from touching the walls. I connected the leads from the power switch to free GPIO pins on the motor server, which extend the GPIO from the RaspberryPi—this is used to send signals to shut down the RaspberryPi.¹⁵
6. Next, I calibrated the voltage step up motor to produce exactly 6V, using a voltage meter and battery pack. I used a spare USB-A cable and cut off one end, exposing the power and ground wires. I connected these to the power terminals on the motor driver, and plugged the USB-A side into the step up module. I attached the

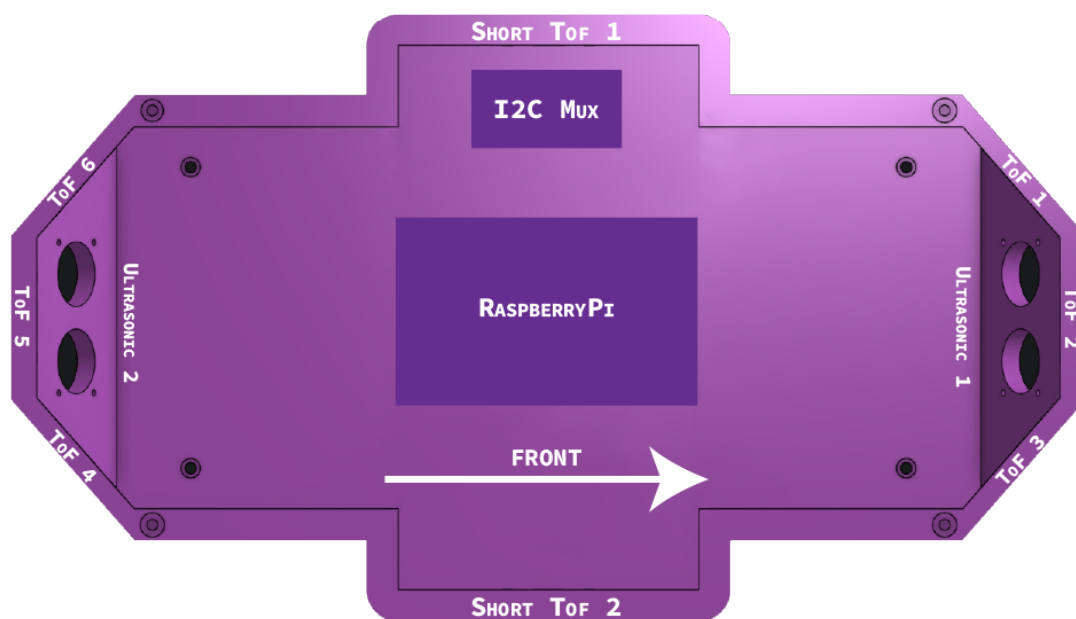


FIG. 10. PyBot V3, sensor placement.

¹² One of my spacers stripped while I was screwing the hex spacers into the main body—I will increase the tolerance for any future prints, but did not reprint to save filament.

¹³ As previously mentioned, I soldered the GPIO header and motor/power connections to this part prior to assembly.

¹⁴ The data wires from the motors may be of use if I need tight calibration of motor speeds, but I removed them from the wiring harness as I don’t plan to use them.

¹⁵ I have not implemented this yet.

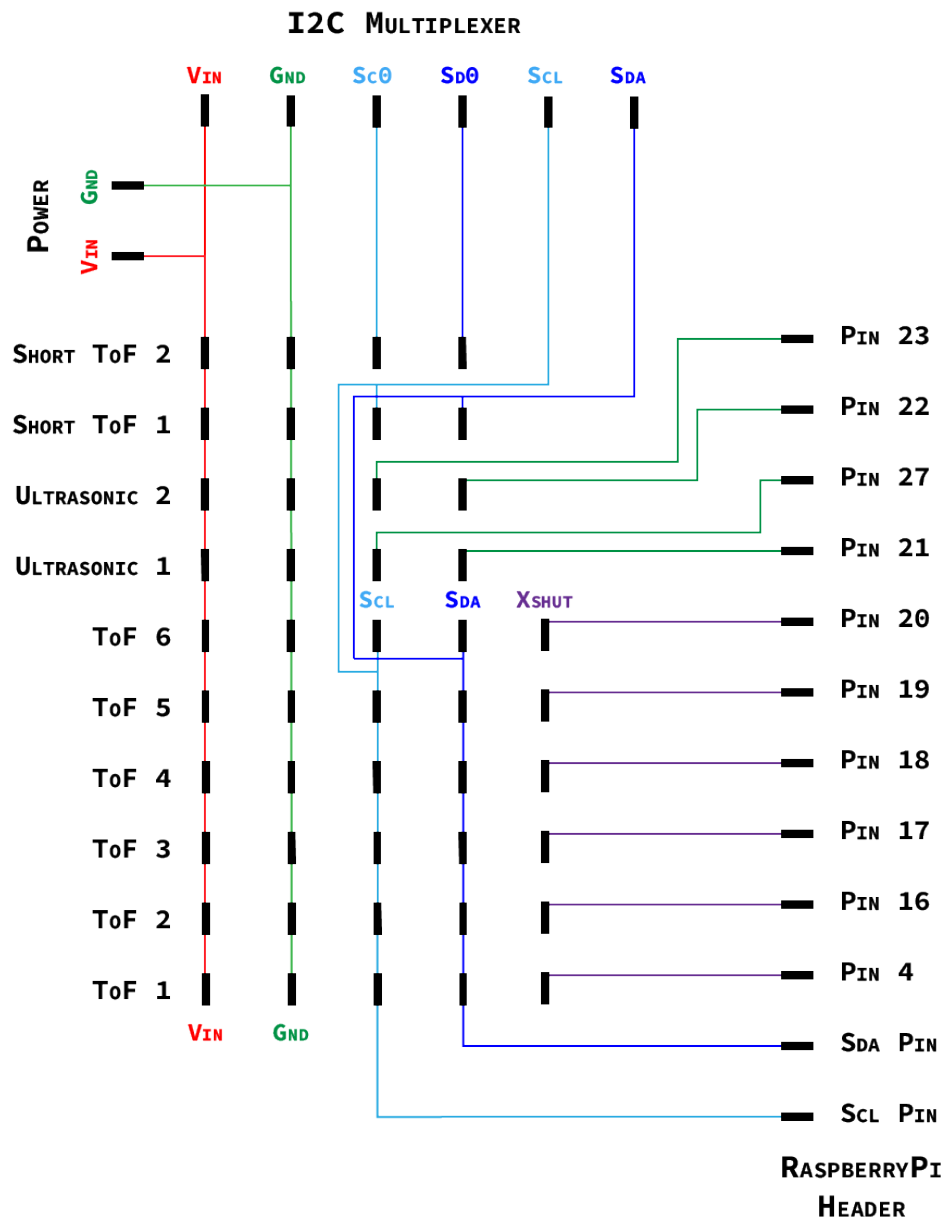


FIG. 11. PyBot V3, sensor wiring diagram.

USB-A extension cable to the other end, to be attached to the battery pack later. I had the step up module sit in the underchassis to the side of the RaspberryPi.

7. I put another hex spacer on another corner of the RaspberryPi and attached the GPS board, then connected it to the RaspberryPi with a USB-C to USB-A male to male cable. The GPS module sits on in the underchassis, on the opposite side of the step up module. The completed underchassis assembly is shown in Figure 12.
8. Before placing the proximity sensors on their mounts, I soldered the headers on each of them.¹⁶ This allows me to access the XHUT pins on the ToF sensors, and lets me be a little clever with my wiring later.
9. With the underchassis complete, I attached my I2C mux to the main chassis, and screwed in the ToF sensors to the board. Some of the screws were a little loose, so I used teflon as necessary. The Ultrasonic sensors fit very snugly in their holes, so no screws were needed to attach them. See Figure 10 to see the placement of the sensors on the chassis.

¹⁶ Note that I did not do this for my short range sensors, as they do not have XSHUT pins.

FIG. 13. PyBot V3, assembled main chassis with battery.

10. With all the wiring of the underchassis complete and the sensors attached to the main body, I placed the main body on the underchassis and screwed them together, using the hex spacers placed into the underchassis earlier.
11. Next, I wired the proximity and ultrasonic sensors together according the diagram in Figure 11. Initially, I attempted to wire the sensors in parallel, but this was very difficult to cable manage and I do not recommend taking this approach. Instead, I attached all of the ToF sensors and one of the two short range ToF sensors together with STEMMA QT cables, so they share I2C and power communication. Then, I used a single STEMMA QT to female sockets cable to connect all of the ToF sensors to the RaspberryPi I2C and power, using the pins on the motor driver. Next, I connected each XSHUT pin to GPIO pins on the RaspberryPi using jumper cables. This allows us to turn off the sensors to change the addresses.
12. For the remaining short ToF sensor, I used a STEMMA QT to female sockets cable to connect the I2C pins (SCL/SDA) to one of the numbered I2C pins on the I2C mux (SC0/SD0). I then wired the power pins to a normal ToF sensor. I also wired a normal ToF sensor to the I2C mux, connecting the I2C mux to power, and to the RaspberryPi I2C bus.
13. Finally, I made four shorter jumper cables with some plyers. I used these short cables to connect from the front/back ToF sensor power headers to the Ultrasonic sensors power headers. I connected the other two Ultrasonic sensor pins to free GPIO pins on the motor driver with normal jumper cables.¹⁷ The assembled main chassis is shown in Figure 13.
14. Finally, I started work on the cover, beginning by screwing the camera to the camera mount.
15. I also attached the touchscreen display to the top of the cover, with the display connector protruding into the cutout, allowing the screw holes to rest flush with the plastic.
16. With the display and camera attached, I attached the 1/4in ball mount, using the 1/4in mounting screw through the cover. I then attached the IR floodlight on the ball mount.
17. Next, I connected the voltage step up module to a USB-C hub.¹⁸ I also connected a micro USB cable to the hub, which plugs into the IR floodlight. I plugged in the RaspberryPi using a right angle USB-C extender and USB-A to USB-C cable and connected the other end to the battery. I also connected the micro USB power cable from the I/O extension, which will be used to charge the battery. Finally, I attached the USB hub and placed the battery in the chassis.
18. With all parts attached, I connected the display and camera cables through the cover. I also attached a GPIO cable from the camera header¹⁹ to pin 25 on the motor driver header.²⁰
19. Finally, I attached the voltage step up USB-A and RaspberryPi power USB-A cables to the battery outputs, and the IO micro USB cable to the battery input. I screwed in the top cover to complete the assembly.

The completed assembly is shown in Figure 14.

¹⁷ Taking time only wire I2C and power to the RaspberryPi from one sensor allows for much better cable management and makes it easier to access the underchassis.

¹⁸ Note that this is not included in Table I, as I did not purchase this part, I had a spare USB-C hub. I recommend buying a USB-A splitter for this purpose. Unfortunately, my model of battery has a low voltage timeout and will not keep powering USB-A at low voltages, the USB-C hub was the only part I had on hand that worked with my battery. The battery listed in Table I might not have this low voltage limitation.

¹⁹ I soldered the header on the camera earlier, the camera arrived with a hole and no instructions.

²⁰ This allows me to turn on the camera night vision with GPIO pins instead of hardcoding the configuration in Raspbian's boot config.



FIG. 14. The assembled PyBot V3.

IV. FUTURE WORK

Some final assembly work remains, namely reprinting the mecanum wheels with slightly lower tolerance to prevent them from occasionally falling off, or finding an alternate way to better secure the wheels to the motors. Additionally, some software development for the web interface remains. I wish to add proximity sensor data, implement a way to set the touchscreen message, add hardware configuration settings, and add some control over the RaspberryPi (for example, power options and system logs). These would make the interface more usable and allow for more remote control over the robot.

Additionally, I did not have time to work on self-driving capabilities. To begin, I will train machine learning models to follow tracks that will be laid out in tape. A human operator will then drive the car on these defined tracks multiple times, recording video and controls for each run. I will use these trials to train the first iteration of the machine learning algorithm.

There are online code repositories that can use camera data to train algorithms to follow tracks. I will not use these libraries as it is not applicable to other goals of this project, but they may speed up this portion of the project.

Most previous work uses neural networks or reinforcement learning for self-driving model cars. Neural networks are the more common and well documented method to train model car to drive along tracks. Implementing this method should not be overly complicated, and I can use existing code repositories to guide the model training for this initial goal. DonkeyCar is the most popular of these libraries and has lots of documentation [3]. If a reinforcement learning algorithm seems like a better option, the research group could use a different GitHub Repository [4]. There are many more repositories and papers detailing training model cars on tracks, but these options are well documented and widely used.

After replicating previous work, I will attempt to train a reinforcement learning model to drive PyBot, without using the camera at all. This type of machine learning can drive PyBot by building an algorithm that is rewarded for achieving desired goals, such as getting closer to a specified destination, and punished for misbehaving, like crashing into a wall. If I train the model from scratch and use the car from the start, the model will not know how to begin. This algorithm behaves like a child; they don't know the stove is hot until they touch it. Thus, I will either begin training the model in a simulation, or start training with open source code that can start with a baseline. If possible, the model will be trained in simulations, as other projects won't have the same sensors and configuration as PyBot.

I intend to make a simplified representation of a building using a 2-D plane for initial training. This will create a representation of real-world driving without an overly complex simulation. There will be an agent (PyBot) that moves around the 2-D representation of a room. In more complex models, the camera simulations may be included, but the project will start out by only simulating the proximity sensor data and controls that the model will have over the agent.

An example of a 2-D plane that I might use to train a reinforcement model is shown in Figure 15.

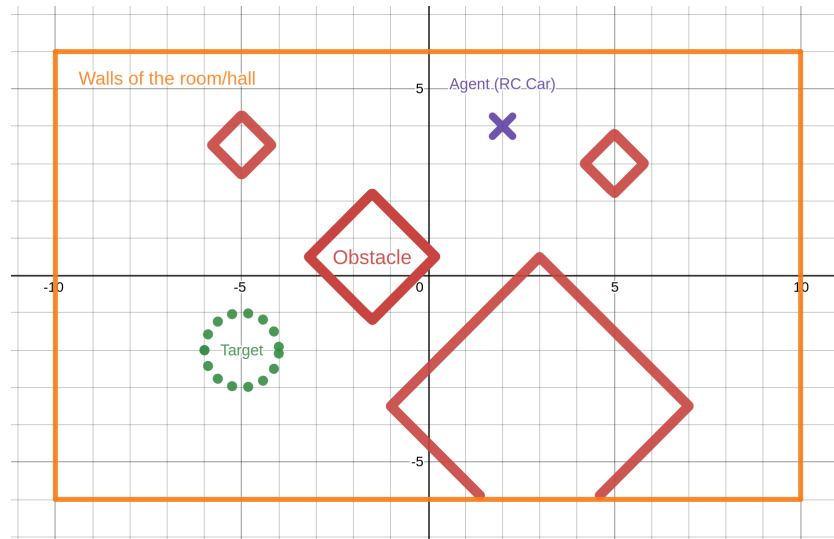


FIG. 15. This is an example of a 2-D plane that could be used to train a model.

In the real world counterpart to the agent, PyBot, I have mounted proximity sensors on each side of the chassis, with extra sensors on the front and back. These sensors will feed the model the distance of the closest obstacle in each direction. In the simulation, the model would calculate the closest intersection along the line of the sensor. Since the real sensors on the car are very narrow, this is an easy calculation in the simulation.

After constructing the agent and testing environment, I can define the reward function for the reinforcement learning model. The algorithm will be rewarded for getting closer to the target (calculated using the GPS in the real world), and punished for getting too close to obstacles. I will need to make iterative adjustments, as reinforcement learning algorithms are notorious for following rules in unexpected ways. For example, if I rewarded the algorithm for reaching the goal and heavily punish it for getting too close to obstacles, the algorithm might find the most rewarding solution is to stay at the start location. Thus, the model should be punished for sitting still too long.

This process will be iterative, as is common for machine learning training. The model will need to train on many environments before it can be trusted enough to be given control of the actual car. The research group could go directly to real-world training, but a simulation can train much faster than the real car, as it is not slowed by needing to physically move motors. The reason we need a simulation is that the algorithm will be very bad at controlling the car at first. The car shouldn't repeatedly run into obstacles and potentially damage itself or the room before it learns not to do that.

I want to keep the model as generalized as possible, so my model may exclude the camera data. Using the camera might overfit the model to a specific environment. This means that the model might perform amazingly in a certain building, but it could fail if used outside or in another building. If I train the model with camera data, the algorithm might use specific features in the testing environment that do not exist outside of it. There are methods to minimize overfitting, but I will avoid using the camera data unless necessary.

If the reinforcement algorithm is successful using only proximity sensors and GPS data, it should be easy to apply the model in different environments. The most difficult environment to test may be outdoors, since the sensors may need to be adjusted to detect sudden height changes—this may be useful anyway, as I don't want PyBot to drive down stairs.

I am also considering investigating self-driving capabilities similar to algorithms used for NPCs in video games. These algorithms are not machine learning, but they can be very effective. I could test such algorithms with the simulations used to train the reinforcement learning model. I have not researched these algorithms in detail, but they may be another avenue to explore.

ACKNOWLEDGMENTS

I would like to thank my advisors for this project, Dr. David G. Fisher for his guidance during my initial iteration of this project in my senior capstone course, Dr. Christopher Kulp for his advice throughout the project and for his role as my Haberberger Fellowship Faculty Advisor and Honors Project Advisor, and Dr. Emily Wilson for her advice and her role as my Honors Project Advisor. I would also like to thank Dr. Krish Pillai and Dr. Andrew Brandon for serving on my departmental honors committee. This work was supported by the Joanne and Arthur Haberberger Fellowship, awarded to the recipient at Lycoming College, Williamsport, PA.

-
- [1] Wikimedia, [Mecanum wheel control principle.svg](#) (2020), accessed: 2023-04-28.
 - [2] Raspberrypi, [Picamera2](#) (2023).
 - [3] Autorope, [Autorope/donkeycar: Open source hardware and software platform to build a small scale self driving car.](#) (2021).
 - [4] J. Dohmen, R. Liessner, C. Friebel, and B. Bäker, in *Proceedings of the 13th International Conference on Agents and Artificial Intelligence - Volume 2: ICAART*, INSTICC (SciTePress, 2021) pp. 1030–1037.



LYCOMING COLLEGE
CENTER FOR ENHANCED
ACADEMIC EXPERIENCES

Haberberger Fellowships