IOT Lab 1: IoT Devices

DISCLAIMER: *This lab looks long - do not worry! Most of this document is more like reading material than a lab. Our goal is to provide you with extensive descriptions, to walk you through each step, so you will not get stuck, and to also teach you additional things. It is ok to skip over parts of the reading if you feel comfortable and don't feel like you need the extra help.*

Lab Overview

Modern cars are no longer just engines and tires. Inside of them are highly sophisticated computer networks, controlling and monitoring an array of sensors, machine learning and computer vision algorithms, performing mathematical computations to keep you and your passengers maximally safe in every contingency and danger that arises.

In this class, we will guide you to gain experience with Internet of Things devices. You will do this by implementing a 2021 Tesla Model S. In particular, you will be

implementing a vehicular network, navigation systems, and computer vision infrastructure comparable to a simplified version of the 2024 Tesla Model S. To save on costs, we won't be building a life-sized car, we'll build something smaller (and simpler!). But don't be fooled though - your infrastructure will share several key capabilities with real car platforms, performing real-time communications within the automobile to perform life-saving features such as obstacle avoidance and lane departure mitigation. Doing this will give you experience in programming IoT components, as well as teach you about vehicular networks, an emerging powerful use case for IoT.

For many of you, this will be your first experience working with real hardware. This may make this lab assignment different from other labs/homeworks you've worked on in the past. A few things to keep in mind (a) Start early in this Lab since things can go wrong when you are interacting with the real world (with an ultrasonic sensor, etc.) (b) if you encounter a problem, try googling for it or searching for posts in the forums - some solutions are already online. Your efforts will be rewarded at the end of the lab with a confidence that you can build real things.

Before implementing vehicular networks on a real car, an important step is prototyping. The common approach in the industry is to test with the intended workflow until we have enough confidence with the structure of the system, before starting to produce the hardware. Although a true vehicular network may contain more components in the architecture, for the purpose of this lab we will take a simplified approach, building one individual car prototype with a simplified degree of autonomy.

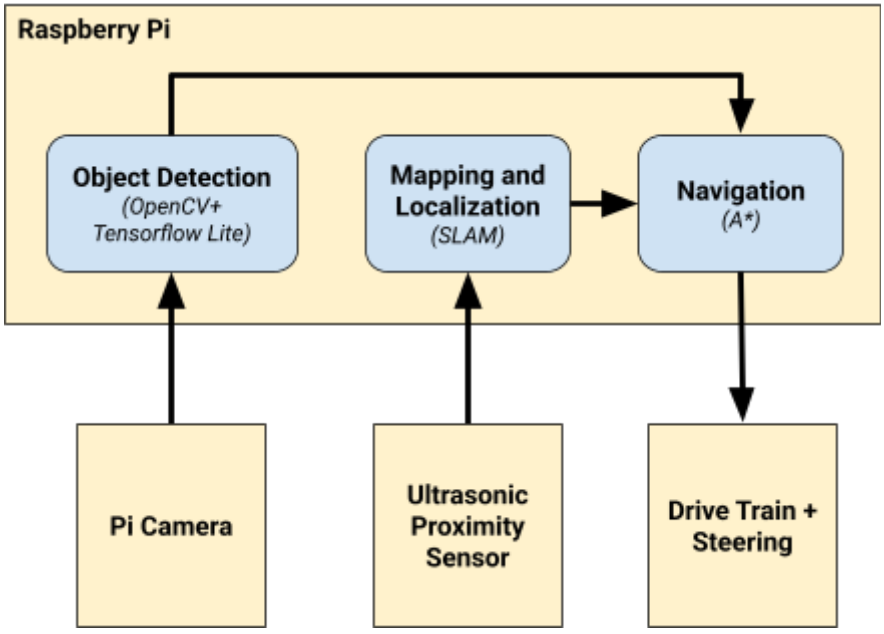


Figure 1: What you will implement, system architecture view

Tasks You Will Perform In This Lab

Part 1:

1. **Assemble the car chassis.**
2. **Explore the code.** Familiarize yourself with the code dealing with the ultrasonic sensor, velocity sensor, etc.
3. **[Submission] Create a demo video and submit a report.** Details of submission and grading rubric can be found in Part 1 Submission and Grading.

Part 2:

Start early on this part as you are required to construct a complicated system with multiple modules.

1. **Construct advanced mapping leveraging proximity detection.** You will leverage the ultrasonic sensor to receive distance data and build an advanced mapping algorithm.
2. **Provide object detection capability to the car.** You will use OpenCV for image preprocessing and TensorFlow for object detection.
3. **Provide routing capability to the car.** Utilizing the advanced mapping algorithm, you will implement a navigation algorithm to route from a given starting point to a goal.
4. **Build a fully self-driving car.** You will integrate all parts from previous tasks and demonstrate that your car is fully self-driving.
5. **[Submission] Create a demo video and submit a report.** Details of submission and grading rubric can be found in Part 2 Submission and Grading.

IOT Lab 1: IoT Devices (Part 1)

Ok, now let's get started! In part 1, you will assemble the car, explore the Pycar code, and implement some naive mapping and routing.

Step 1: Chassis Assembly

A key step when constructing an IoT device is putting together its housing and physical assembly. Oftentimes this will be the last step in your design process, after you have fully thought through about the structure of what you want to build, its environment, and so on. But in this case, since we know what we're building, we'll start with this step.

If you have not already done so, make sure you did lab 0, which overviews what parts to purchase.

To put together the car housing (chassis), as well as setting up the Raspberry Pi to function with it[1], please follow the guide found at this link. (backup link: https://m.media-amazon.com/images/I/C1Tq1JjfipS.pdf)[2]. You may also find these youtube videos helpful:

> https://www.youtube.com/watch?v=7m1DLLG3A8s
> (10 min tutorial on assembling the Picar)
>
> https://www.youtube.com/watch?v=UiRb1SDpezY
> (2 min video showing (tracking line, objection detection, keyboard controls)

The link above will walk you through how to install the Raspberry Pi operating system on it, but if you could use some additional instructions you can follow these:

---

[1] One thing to be careful about is to ensure you have enough electrical power. E.g., if the batteries get low, the picar lights might power up but the Pi might not have enough power to boot. Make sure the batteries always have a full charge. In fact, while you're developing, you might want to have the Pi connected to an external power source (any USB C power source 15W or higher should work).
[2] One small suggestion is you may want to pause at page 45 of this manual to make sure the Pi Camera module is installed, otherwise you'll have to disassemble the hat and reinstall it, which might slow you down a bit.

https://docs.sunfounder.com/projects/picar-4wd/en/latest/preparation/installing_the_os.html

After you have flashed the image, insert the card into Raspberry Pi, and connect a monitor, mouse and a keyboard to the Pi via the HDMI and USB ports. The Raspbian OS would automatically be installed once you start the Pi. Now you will have a fully functional Linux system on your Pi, with Internet access via the Wi-Fi module. Note you don't have to specifically use the optional power adapter mentioned early, any USB-C connection with enough amps will keep the Pi powered on (it will give you warnings or not power on if it's not getting enough power).

To assist you through this process, we'll walk you through in more detail the steps of setting up the Raspberry Pi next.

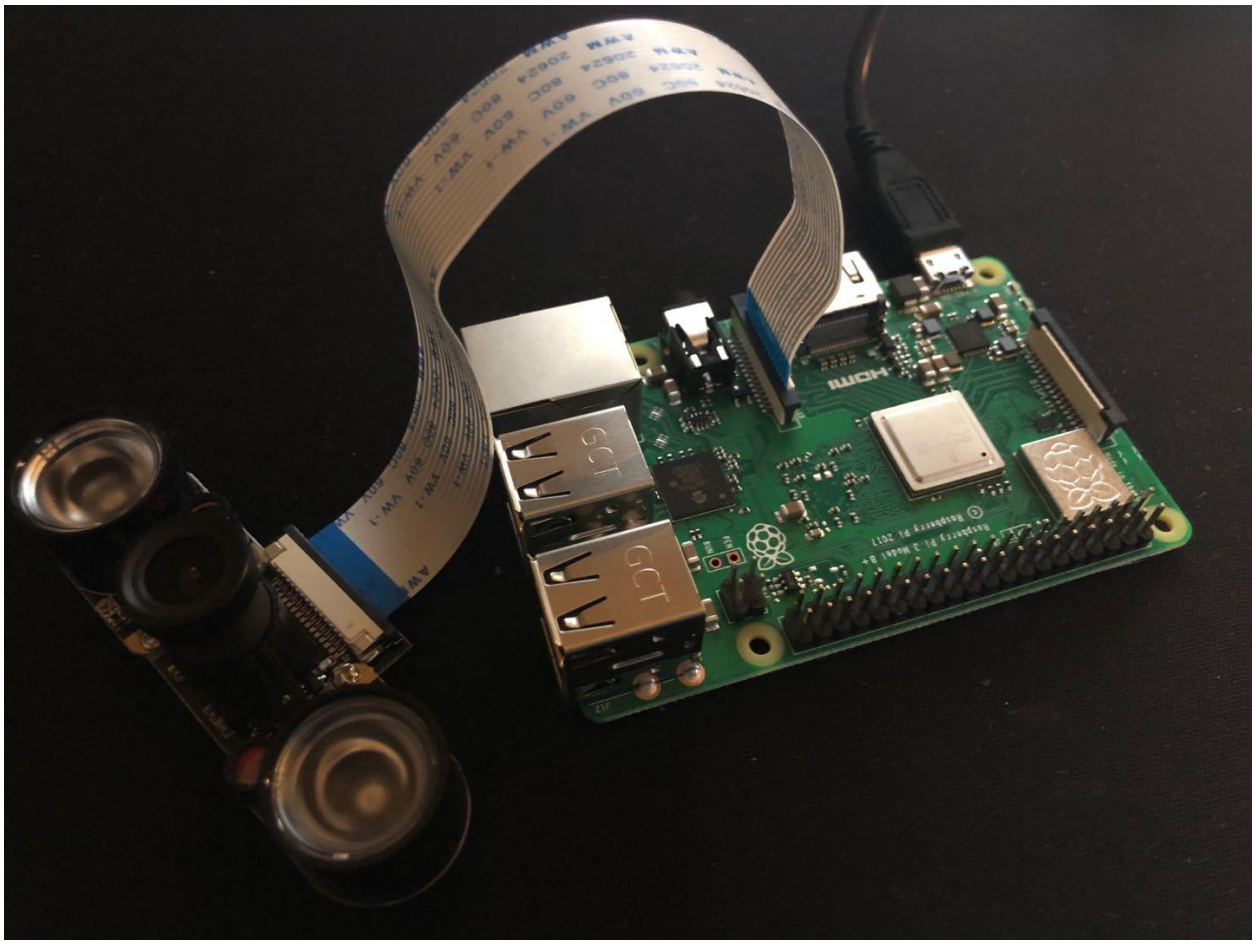


Figure 2: Raspberry Pi with Camera installed.

Do not underestimate a Raspberry Pi - It's cheap (~$40), but it's a powerful computer for its size! Equipped with a Linux Distro (Raspbian), an HDMI port for Monitors, USB ports for mice and keyboards, an SD card as a disk, a camera interface, Wifi and Ethernet module, and 40 GPIO pins, Raspberry Pi allows you to quickly prototype your IoT network without much hassle. When you have set up the Raspberry Pi, you will have a fully functional Linux system on your Pi, with Internet access via the Wi-Fi module. We will be using it to control the car in various ways, but please keep it forever, as it can do so many amazing things beyond just what is described in this lab!

Helpful hints:

1. If you use a 64 GB microSD (or larger), please follow the instructions in this link to format the microSD with FAT32[3], instead of the step 2 in page 9 in the above guide. This is necessary because the SD Formatter will use exFAT which is not supported by the Raspberry bootloader.
2. The Raspberry Pi 4 has an HDMI port that you can use to connect it to a monitor if you have one[4]. If you do not, Pages 20-21 of the above guide tells you how to install the Pi up in "headless" mode. Those instructions are ok, but here are easier ones:
    a. First, download the imager from https://www.raspberrypi.org/downloads/
    b. Then, run the imager. Follow the steps it provides to put the recommended OS onto your SD card.
    c. The SD card should now have the OS - add an empty `ssh` file to the root of the SD card to enable ssh access.
    d. Add a wpa_supplicant.conf file to the root[5] of the SD card using the following instructions: https://www.raspberrypi.org/documentation/configuration/wireless/wireless-cli.md . Change the country code from US to your 2-letter ISO 3166-1 country code if not in the USA. Also change the `ssid` (name of your Wifi network - note this is case sensitive) and `psk` (your Wifi password) Sample file:

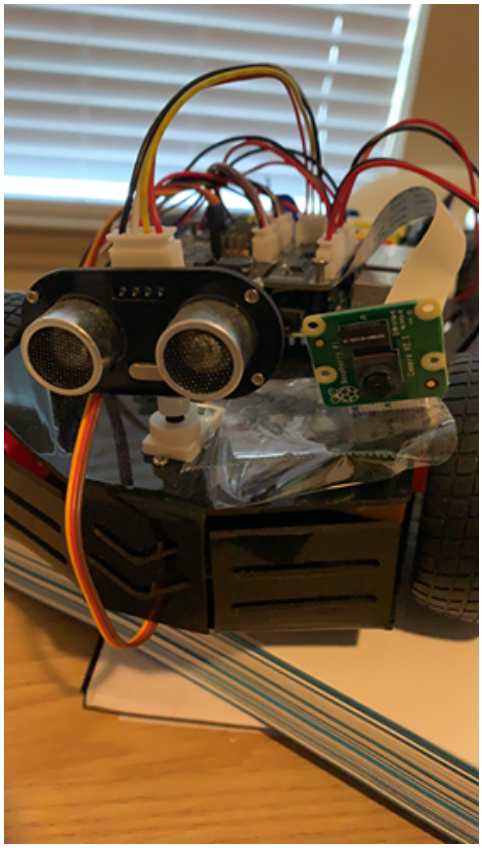
[3] https://ragnyll.gitlab.io/2018/05/22/format-a-sd-card-to-fat-32linux.html
[4] Note you don't have to do this -- you can also use VNC to connect over SSH, as described above.
[5] If these files disappear from here after reboot - don't worry, that is to be expected. See: https://raspberrypi.stackexchange.com/questions/68808/raspberry-pi-zero-w-keeps-deleting-wpa-supplicant-conf-and-ssh-file . You may want to make a backup of these files elsewhere in case you want to edit and make further changes on them later.

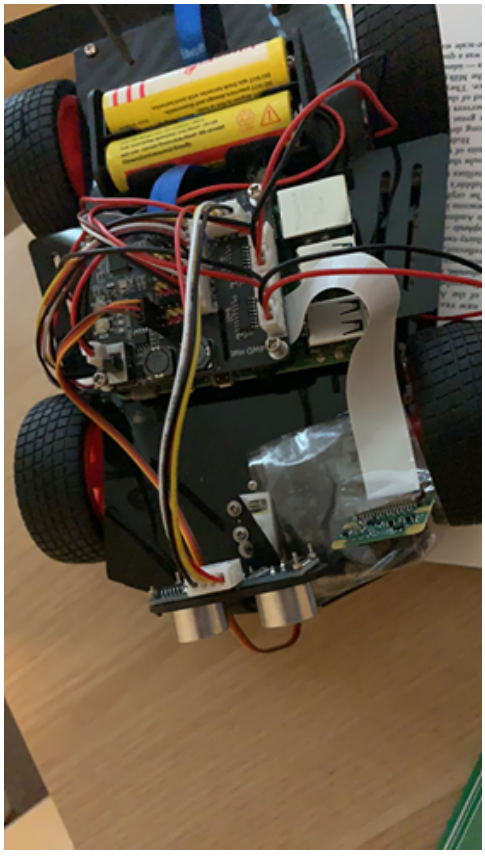
```
ctrl_interface=DIR=/var/run/wpa_supplicant
GROUP=netdev
update_config=1
country=US

network={
      ssid="WiFiName"
      psk="WiFiPassword"
      id_str="school"
      priority=0
}
```

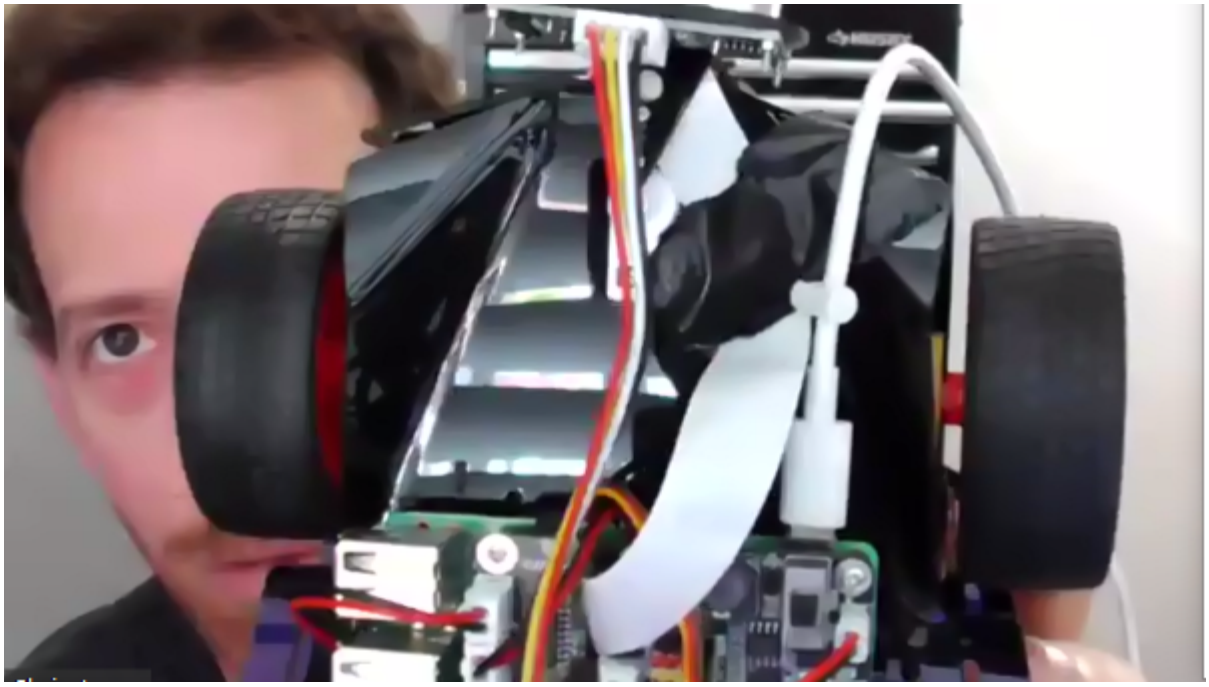
For our purposes, we will need to use a camera module, which is not part of the instructions in the link above. Formal mounting of the camera should not be necessary; folding the camera to face straight forward and applying tape would be sufficient. If you would like something a bit more solid, you can cut a slit on top of the PiCar and mount the camera through that slit[6]. To ensure you have the camera mounted and oriented properly, please load a picture on your cell phone's screen, and hold it in front of the car where you think objects might reasonably appear when the car is driving around. Then, look at the frame of the camera and place a picture on your phone within the frame of the camera and see if your car detects it and reacts as expected, i.e. stopping and waiting for the person to pass. If needed, adjust the camera placement on top of the car so the car can "see" the road nice and well.

Here are a few screenshots of the assembled car from a few different angles -- take a look and do some sanity checks to make sure you assembled it properly:

[6] Or if you want something even tougher, maybe you're planning on raising the suspension and taking this thing offroad, you can drill holes to mount the camera with 4 bolts/nuts. Alternatively, you can flip the camera over and trim the plastic at the edge of the flange from the top piece of the car's frame and slide the ribbon cable through and then cut another slot to feed the ribbon up to the RPi. If you choose zip ties and you pull the zip too tight it'll aim it too far down to the ground - you can wedge something behind the camera so that the connector is flush with the mount. But none of this is really necessary,







The Raspberry Pi should be set up first, but after that, the chassis assembly and the routing/mapping logic (we'll get to these in later sections) can be done in parallel. Note that the picar-4wd library code includes functions and example scripts; use them as you see fit, and feel free to fiddle with the car's config file for car motion.

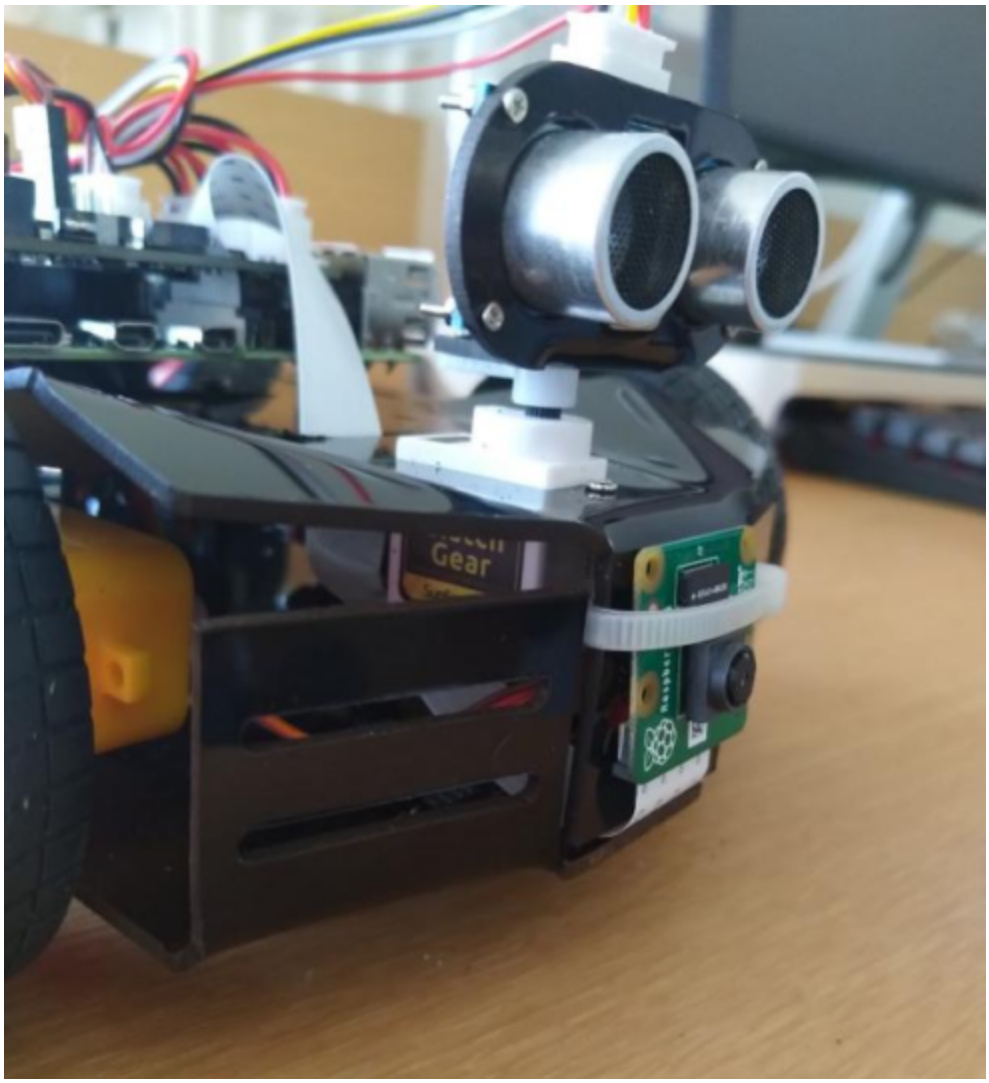


Figure 3: Mounting the ultrasonic sensor atop your smart car.

There are example scripts at the bottom of the setup pdf (keyboard_control.py, servo, etc) that can be run to ensure you're assembling the chassis properly.

Step 2: Implement your design on a PCB

Note: *Don't worry about getting this step perfect. The goal is to give you general experience with PCB design in case you need it for the future. Go through the steps and do your best. You should generally be able to get it working. If there is a box here and there that is a bit off we won't take off points for that. Get it generally working and you will get full credit.*

Printed circuit boards (PCBs) are a massive part of just about every piece of electronic equipment that we have in our everyday lives. Almost every electronic device you can think of contains at least one PCB, including phones, computers, tablets, televisions, cars. Because PCBs utilize copper tracks rather than actual wires, boards are smaller and they are not as bulky, power usage is vastly reduced, reliability is improved, etc. Hence when you build real IoT devices you don't just connect wires into sockets like you've been doing, you might do that for prototyping, but to do things for real you build PCBs. In general IoT systems, most of the devices have at least one PCB in it. If the whole system is designed to become one product, such as smart cars, a large PCB might be in it to connect to all other components inside it. Using the PCB in devices or the whole system can make them smaller and more portable. The PCBs are also very durable and long-lasting. They can take a lot of damage such as heat, moisture, or even physical force, without breaking apart. This makes them ideal for use in areas that are hazardous to electronics. In addition, PCBs are very efficient and easy to repair.

So how is a PCB physically created? A PCB is an electronic board with metal circuits embedded in it that connect different components on the device. The mechanical structure is made up of an insulating material laminated between layers of conductors. These layers form the circuit board and connect to components that are soldered to the board. With this multi-layer design, PCB can connect to multiple components on a small board. That is why it is very popular in the industry these days.

In this step, you will gain experience in using commercial-grade software[7] to create a PCB design for your car platform. In particular, we want to build a PCB to connect all your components and raspberry PI together, which is similar to the role of the 4wd board. Rather than build another 4wd board, we want to build our own board that ONLY connects to our components and raspberry PI. For the submitted requirement, you need to include at least **motors, photo-interrupter, battery and grayscale module**. If you cannot find the footprint for your component, you can use different types of the same component, such as different types of battery holders in your design. You can also put a connector header on your PCB and mark the

[7] Please consider to list some of the skills you acquire in this class on your resume, if you feel they could fit. KiCAD is a widely-used software in industry, and the experience you gain here will be transferable to other commonly used platforms like Altium Designer.

component it connects to aside it. You need to use a line to address which component link to which header in your design. We suggest you design it from the raspberry PI hat, which has basic pins that connect to raspberry PI.

As many PCB design tools can be found online, we recommend you use KiCAD, a free and open source PCB builder. KiCAD also has very active and growing community of users and contributors. Last but not the least, KiCAD is cross-platform compatible, which means it has Windows, Mac and Linux versions for different users. You can download KiCAD here. you can go through tutorials like this, associated with the official document, which helps you make a clear understanding on how to build a PCB on KiCAD. We also provide some steps to start with KiCAD:

After you download KiCAD, we can start a new project in KiCAD. Open a new project from the template. Choose the Raspberry PI HAT from system templates (Figure 4). Press OK to start the project.

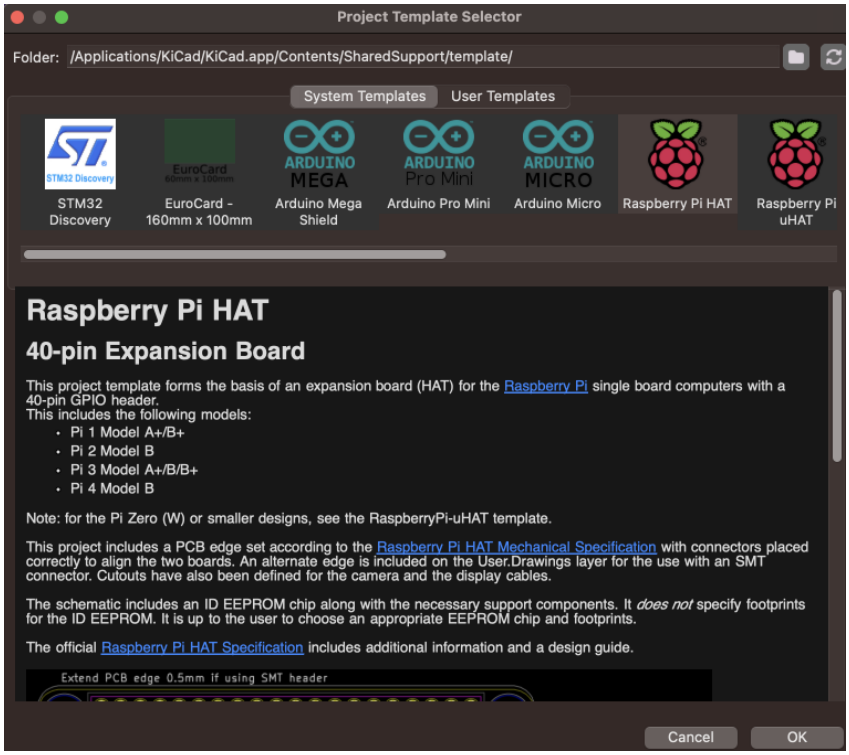


Figure 4: Open a Raspberry PI HAT project on KiCAD to start building PCB

Next, open the schematic editor and import the footprints of all your components. This is the place where you see all the pins and wires in your PCB. You need to add all your components' footprint in it and create pins on board to connect to it. You can download the footprints for your components on this website. A footprint defines the arrangement of pads and pins used to physically attach and electrically connect a component to a PCB. The PCB footprints usually include information such as copper and solder mask layout, silkscreen, mounting holes (where applicable) and reference pin. After you download the footprints, press A to add symbols that

represent your components in the schematic editor. Wired all symbols together. Don't forget to run the footprint assignment tool on the top toolbar to assign each of the symbols you insert into the schematic editor to it's footprint. Figure 5 gives an example of the ultrasonic symbol wired with a 4-pin connector.

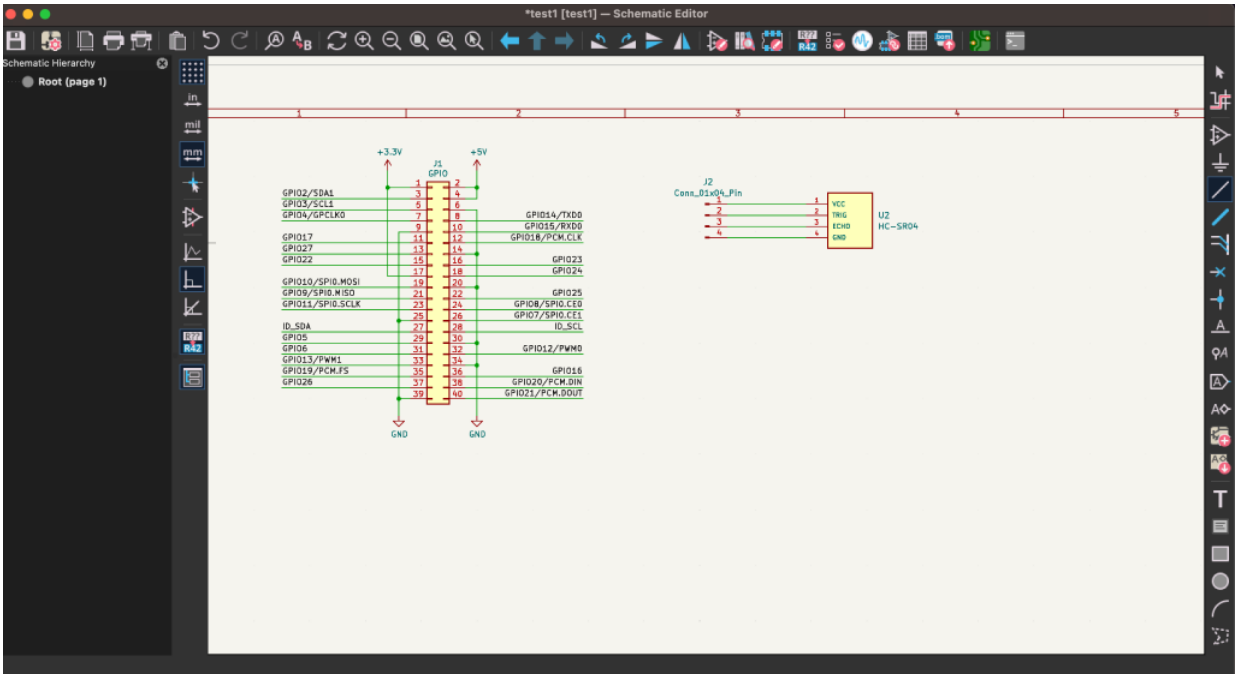


Figure 5: An example of ultrasonic wired with 4-pin connector

After finishing editing symbols and wired them together, you should have wired all components to the board. Next step is to arrange and design your board with components on it. Open the PCB editor. This is where you can see the board. In the upper toolbar, use the tool "Update PCB with changes made to schematic". This will make all your components and wires you put in the schematic editor show in PCB editor. Rearrange all components to make sure there is no overlap between components and wires.

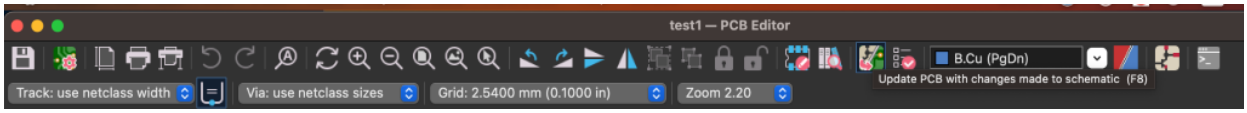


Figure 6: The "Update PCB with changes made to schematic" button.

Finally, select "File" -> "Plot…" in the PCB editor. We want you to generate gerber files for us to view your result. The Gerber format is an open, ASCII, vector format for printed circuit board (PCB) designs. It is also an industrial-standard PCB describe file. By providing this file we can see your PCB in KiCAD. Check "F.Mask", "B.Mask" and "Edge.Cuts" and leave all other options as they are. Click "Plot". Then select "Generate Drill Files…" then click "Generate Drill File" without changing anything. We now have each layer of our PCB into a gerber file. Close gerber

file generator and PCB editor.

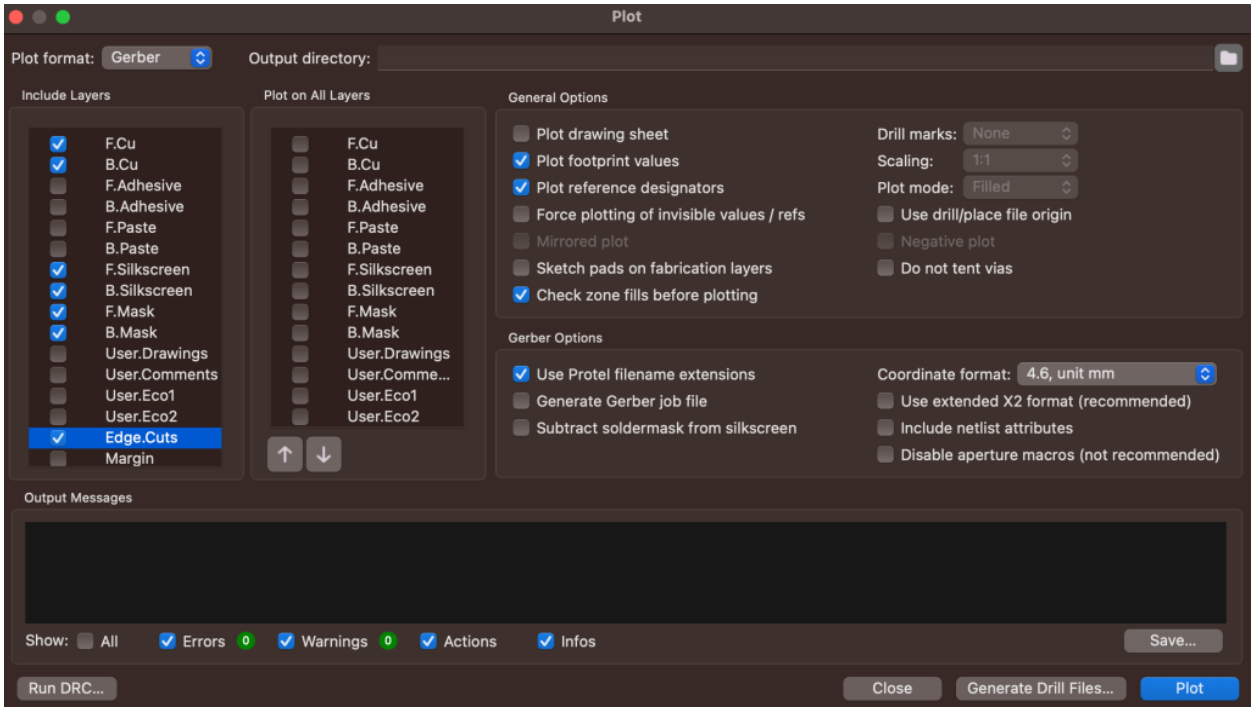


Figure 7: The checkbox of the Gerber generator you need to check to get all layers of Gerber files.

Check the gerber files by opening Gerber Viewer before you submit the gerber files. In the viewer, select “File” -> “Open Autodetected Files” and choose all gerber files you just created. Check each layer of your gerber files to make sure your wiring design and components layout are all correct. Figure 8 shows an example on how the Gerber files can be looked like.

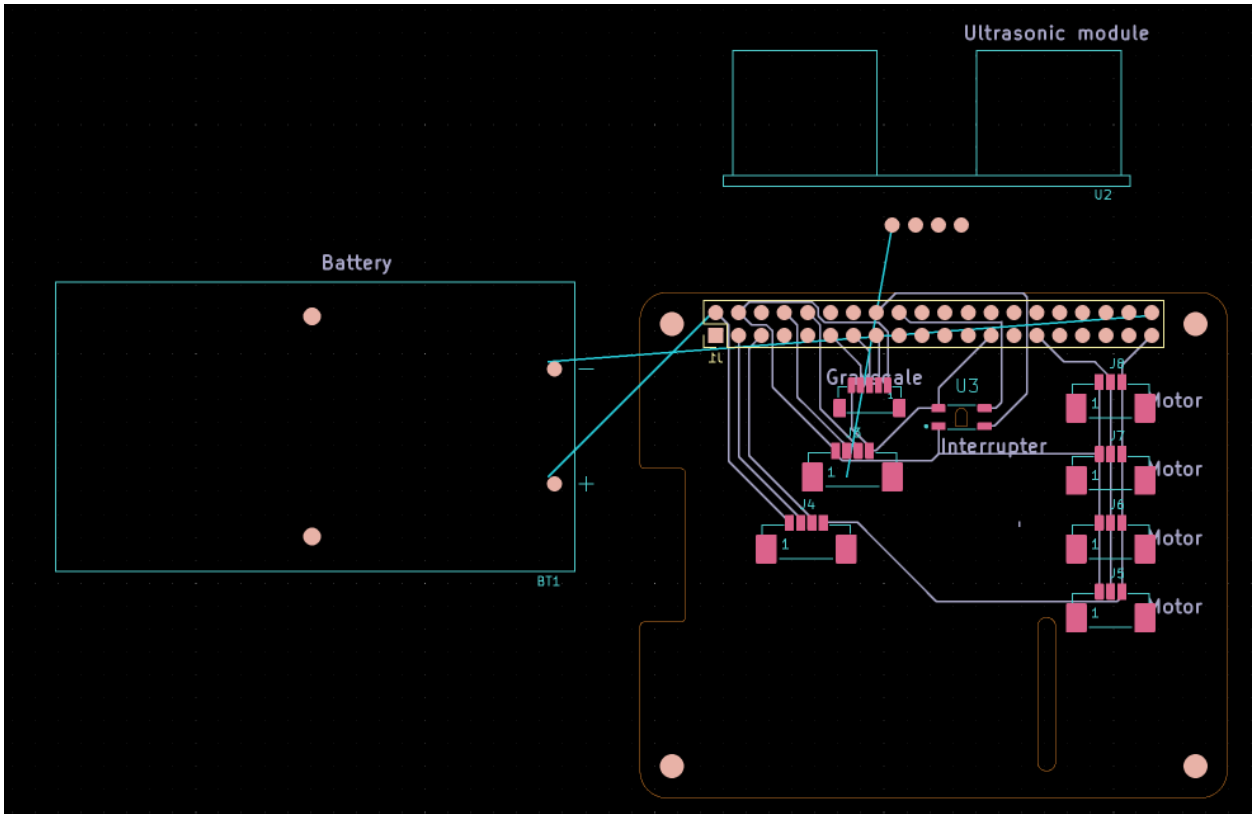


Figure 8: An example of the Gerber files on final PCB design.

Now you have finished your PCB design. Usually the next step is to print out your PCB[8] and buy all components, then assemble them. Since we already have Pycar, we don't need to print our PCB out. But it is still good to practice it before next time you need a new PCB board somewhere else.

Step 3: Explore the Pycar Code

[8] Feel free to actually do this if you like -- there are websites you can go to where you can upload your Gerber files, pay like $30USD, and they'll print your design on a PCB and ship it to you. If you would like to commercialize the project you do in this class, this is a good way to go (they will charge you much less if you're ordering large quantities).

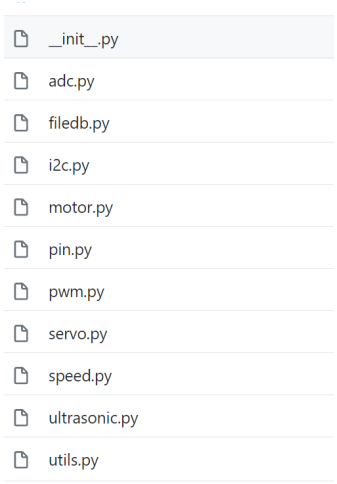


Figure 9: Example code provided with the Pycar library . Look through it and get a sense of what it does!

Now that you have built your car, it's time to test it out and get familiar with the PiCar library (which can be downloaded here , see API documentation here). Before we get into too much programming, we think it will be helpful for you to kind of look over the code and get a sense of what it does and how it works.

The first step will be to download the code[9]. Make a copy of it on your Raspberry Pi and read over the documentation and codebase. You don't need to read everything, just skim over parts and get a sense of how things work. In particular, look under the doc folder with all the markdowns of each function and its descriptions. Or you can take a look at the code in the picar_4wd folder of the repository, particularly init.py. That should contain all functions you need to complete the lab. We encourage you to also write your own sample script and try calling some of these functions, i.e. reading from the ultrasonic or making the motors spin. Play around with the functions and try making the car do different things.

Another helpful part of the codebase we strongly encourage you to read and run, to both verify that your assembly is correct and to get you comfortable with the library, are the examples. The various examples should thoroughly test the functionalities of the car and provide you with some reference when writing your own control script.

First, try running the `keyboard_control.py` program and making sure it works. This is important since it tests all your motors are running in the right direction; if that is not

[9] If you're not so familiar with python, consider watching an online tutorial on how to run a python interpreter.

the case, you could either take apart the car and move around the motors (a rather arduous process) or adjust the directions of the motors in the data/config file. We recommend you also walk through the code and make sure you understand the function calls being made in this script, and see if you can trace the function calls back up inside the code as far as you can, since the library offers us wrappers to manipulate the components. For example, when you call `fc.forward`, that actually leads to the actual outputs in the motors' corresponding GPIO pins to be changed -- see if you can look through the code and understand how that happens.

The next script you should explore is `ultrasonic.py`, which is the script responsible for interfacing with the car's ultrasonic sensor for distance measurements. More information on the ultrasonic module for the Raspberry Pi can be found here, but essentially what it does is send out a small pulse of sound and measure the time it takes for that sound to bounce off the object in front of it and return (similar to what bats do), and then uses that time and the speed of the sound to calculate distance. A lot of this is abstracted away for you, but it's still good to know how it works. Look through the `ultrasonic.py` file and trace back the function calls as far as possible, and get familiar with measuring distance since you will need it in the mapping portion of your script.

The third component worth exploring is the servo, which is what the ultrasonic sensor is mounted on. The servo is a type of motor that allows you to precisely control its rotation, which makes it the perfect option for us to rotate our ultrasonic sensor when taking distance measurements. `servo.py` contains all the wrapper code for interfacing with the servo motor and setting the angles, but again, please trace the function calls and see how the actual component is manipulated. Experiment with these different scripts to get familiar with making the right function calls.

Once you have some comfort with the PiCar library and the abstractions it provides, you can begin writing your scripts! Some good first steps, to see how these different functions can fit together: try having your car go forward by a certain distance, or have your car drive forward until it detects an object within 10 cm of it, or perhaps even try having your car go in a circle. As an example, here is some code that, when called, moves the car backward (or forward, depending on how your motors are configured) by a constant amount. Your results may vary depending on the surface on which this movement is being tested.exactly 2.5 cm.

```
    def move25():
        speed4 = Speed(25)
        speed4.start()
        # time.sleep(2)
        fc.backward(100)
        x = 0
        for i in range(1):
            time.sleep(0.1)
            speed = speed4()
            x += speed * 0.1
            print("%smm/s"%speed)
        print("%smm"%x)
        speed4.deinit()
        fc.stop()
```

Figure 10: Example code to move the car using the picar-4wd library

These are just some example ideas that you can try to get comfortable with the library. When you feel ready, proceed to the next section to get started writing a script to make your car autonomous!

Step 4: Environment Scanning (Mapping)

As an autonomous car drives, it must be aware of its environment. You'd think it could just follow GPS, but driving is much more complex than just knowing what roads to follow. There are other cars on the road, there will be obstacles such as road construction, pedestrians will step out in front, etc. Your car needs to be aware of its microenvironment to perform operations such as crash avoidance, summon[10], etc.

Luckily, the problem of automatically discovering a physical environment is one that has been long studied in the field of robotics. The field of *robotic mapping* has its roots in cartography and computer vision - there are a wide variety of algorithms that have been proposed, many of which typically involve a scanning process where the device "looks

[10] https://www.cnn.com/2019/10/10/tech/tesla-smart-summon/index.html

around" in some fashion, then internally constructs some sort of data structure representation of its environment that is amenable to processing (for later navigation and planning steps).

Real world self-driving cars use similar approaches, leveraging arrays of sensors mounted in various locations on the car, but join this information together with vast amounts of mapping data collected by GPS/fleets of other cars. This is what they use for high level navigation and routing from point A to point B. They also have object detection systems that are pre-trained to detect humans, other cars, stop signs, etc.

Since GPS is not granular enough to support true autonomy; cars make use of probabilistic SLAM (simultaneous localization and mapping) algorithms constantly being fed sensor data, they can pinpoint where they are relative to the map in real-time, and stitch together a thorough map built off pre-existing data and real-time data. SLAM algorithms aim to answer the question “what is the probability of the current map state (described by the locations of the obstacles and the agent) given the current set of sensor observations”. The process of mapping involves computing the probability of each position being occupied whereas localization involves finding the agent’s own frame of reference.

For the purposes of this lab we'll keep things much simpler though. What you will do is write a mapping algorithm that simply keeps your car from crashing into things. Your car will act much like a Roomba - it will approach objects, but before crashing into them, back up and traverse in another direction.

What you need to do: Write a program that uses the ultrasonic sensor to detect obstacles that come within several centimeters of your car's front bumper. When your car gets within that obstacle, it should stop, choose another random direction, back up and turn, and then move forward in the new direction.

Where you place your ultrasonic sensor can make a difference[11], especially if you're interested in exploring more advanced mapping algorithms -- if you're having trouble getting good readings (eg you see noise in them), try to remove possible sources of interference - try putting the car on a different surface (e.g., different kind of floor), take it inside if you're outside, etc. You can also make little blinders that direct the line of vision directly forward, like this:

[11] If you'd like to see further, consider elevating your sensor, for example using an attachment like this: https://www.robotshop.com/media/catalog/product/cache/image/1350x/9df78eab33525d08d6e5fb8d27136e95/w/a/waveshare-jetracer-ai-racing-robot-powered-by-jetson-nano-not-included-6.jpg .

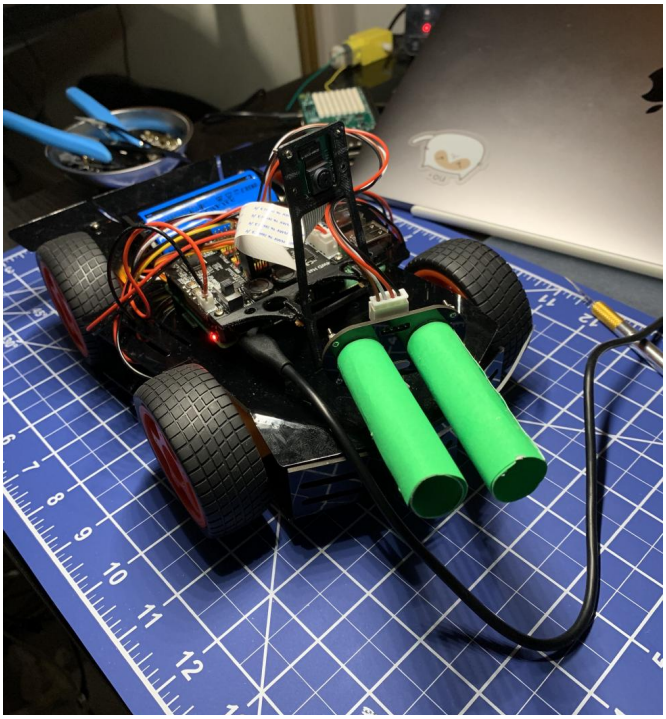


Figure 11: Optional "Blinders" installed on ultrasonic sensor to improve proximity readings (could use dime rolls, rolled up pieces of paper, or 3D printed materials). You don't really need to do this but you can if you want to get extra precision in your readings.

Step 5: Driving

Next, we need to figure out if your car works. One idea would be to take it out on city streets. Put that sucker down in the middle of the road in front of your house and tell it to drive to the grocery store. This would clearly not be a good idea because your car is much smaller than other cars and it would get run over and broken by them[12].

Hence, in this part, you will be setting up a small obstacle course indoors. Pick a room in your house and set up a pretend environment for your car. We suggest something along these lines:

[12] Note it would be very much in your capabilities to attempt something along these lines however - GPS units are available for sale which can be plugged into and read from your Raspberry Pi, you can download street map data from http://openstreetmap.org, and you could probably figure out how to adapt your navigation functions to routing along city streets using that information while avoiding obstacles. To go even further, you could use Openpilot (http://comma.ai - https://github.com/commaai/openpilot) to provide more comprehensive navigational capabilities, or even modify the actual car or truck you drive every day (requires specialized knowledge) with the needed robotics and sensors to make it react to inputs from your software (https://www.youtube.com/watch?v=w8PR5wKT9VA).

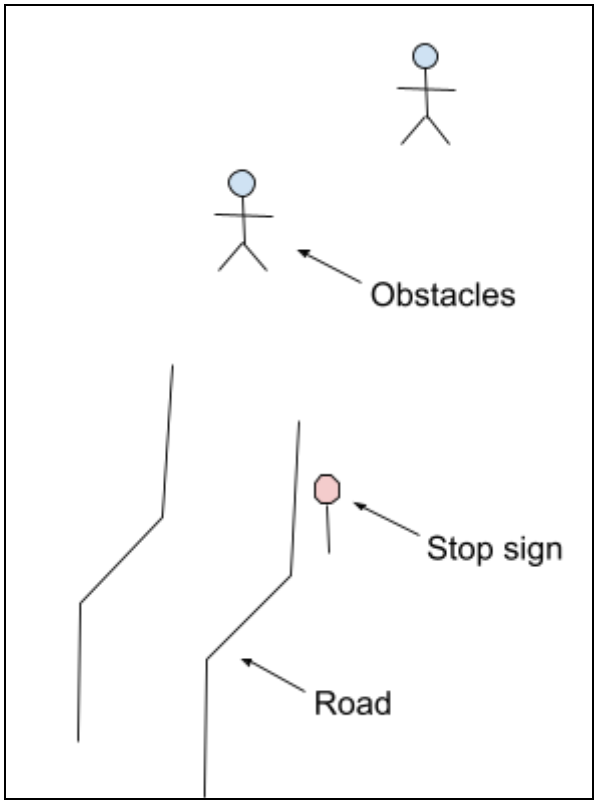


Figure 12: Example environmental setup you can use to test your car.

As part of your obstacle course, we need to test out the avoidance system you developed: Is your car able to recognize objects it should avoid? Is it able to do that accurately? Are there false positives (where it sees objects that do not exist) or false negatives (where it misses objects that are there)? Is your car able to route around objects? Is it able to do so correctly, leaving sufficient clearance around them? Is it able to do so quickly enough, to avoid a crash?

For the routing, the car should go around objects that are obstructing its path. In particular, the distance sensor should be used to recognize that an obstacle is in the way, and halt your car and drive around the obstacle. For an added challenge, try moving your objects around -- routing around the dynamic objects is risky[13] as the objects' movement is far more unpredictable than a fixed obstacle, so we use the distance sensor to act as an override for our routing.

[13] If you think about it, there are actually a lot of tricky issues in routing. Suppose your car is about to crash into a pedestrian, but if you swerve, you'll crash into a building, hurting the driver - should your car choose to hurt you or hurt the pedestrian? Explore these ideas more here: https://www.moralmachine.net/ .

Part 1 Submission and Grading

To submit your own work, you will create a **demo video** (under 5 mins). The video should be hosted privately on cloud, with access permission given to the course staff. **Do not upload the video to coursera**. The demo video should contain:

1) 4 main points in the rubric.
2) The code you wrote as well as a walk through

Your final submission should be a **report** using this template in PDF format**.** The report must contain:

1) The link for the demo video
2) PCB gerber files that include motors, interrupter, grayscale module and battery.
3) Your design considerations for each step of the lab
    - Please also include screenshots of the PCB design and the assembled car in your report

The rubric for grading that you should follow is provided here:

Demo Video (50 pts)	
Is the car fully assembled? (10 pts)	
The hardware connection is shown and discussed (video or report), and the components can power on (Please do a quick walkthrough of the hardware components in video or report.)	10
Things generally seem plugged in but there are clearly a few things not plugged in right or things seem incomplete	5
Can the car move by itself? (10 pts)	
The motors turn the wheels, can the car move under its own power, all without human involvement	10
Movement seems substantially broken in some fashion, e.g., the car attempts to move but the movement is severely hampered, or not all four wheels are turning, or the car is unable to move in a straight line, or if movement requires manual intervention to work	6

Movement does not work at all but students show the car should be able to move by showing their code, and the code seems right	3
Can the car "see" (10 pts)	
The car is clearly able to perceive its environment. It stops when it reaches an obstacle, or otherwise provides some evidence of reaction from sensing its outside world via its ultrasonic sensors.	10
Sensing works but is significantly broken, e.g., it only works sometimes or requires manual intervention for sensing to happen.	6
Sensing does not seem to work at all, but students show the car should be able to sense its environment, by showing their code, and their code seems right.	3
Can the car "navigate"? (10 pts)	
Upon encountering an obstacle, the car can navigate around the obstacle in some fashion, for example by backing up and going in a different direction (it is ok if students did something more fancy here, but in general, the car should avoid obstacles in some fashion).	10
The car is doing some form of navigation but incomplete, e.g., it crashes into objects or scrapes them, doesn't stop quickly enough	6
Navigation doesn't seem to work at all, but students show correct code and pin-point possible issues	3
Code Walkthrough (10 pts)	
Students clearly show and explain the code they wrote	10
Students explain the code without showing, or show the code without explaining, or show and explain only part of the code	0-5
Report (30 pts)	
PCB design (10 pts)	
Gerber files with the required components in PCB design	10

PCB has missing component (-2 pts for each) or the connections are unintelligible	0-8
Report Comprehensiveness (20 pts)	
Report contains detailed information about the students' thought process, such as how they approach the problem, how they reach their final implementation, the challenges they faced, etc.	20
Report is not comprehensive enough	0-19
Submission Format (5 pts)	
Report is submitted in PDF format, video is hosted privately on cloud with access permission granted and does not exceed the 5 minutes limit.	5

Category	**Points**
Video	50
Report	30
Submission Format	5
Total	**85**

Frequently Asked Questions

1. "Can the car move by itself?": The car can move by using the example code "keyboard_control.py". Do we need to write our own code to make the car move or just show in the video that the car can be driven by that example code?
   Ans: It is ok to use that file, just clearly state you are using that code.
2. "Can the car see"/"can the car navigate": I see the example code "obstacle_avoidance.py" and "track_line" can implement the similar functions, but cannot work well, (for example the "obstacle_avoidance.py" can lead to avoiding the obstacle to some extend, but many times the car still crashes into the objects). I was wondering if our code work could base on the example codes and do some improvements.
   Ans: It is ok to base your code on the original code, but go through every line to make sure you understand it (I suggest you write it yourself to force yourself to do that). Also state clearly what code you leverage in your writeup.

3. Topology: I feel for lab1 we don't need to have extra circuits, just using the default assembly (per the assembly instructions) would make it work. So the topology is just shown what it originally has would be okay?
   Ans: That is fine, but please feel free to innovate and create extra circuits, we may give extra credit for that sort of thing.
4. Does one group only need to submit one video and report?
   Ans: One group only needs to submit one video and report, up to the team to decide how to allocate the work among yourselves.
5. How to set the default servo angle to 0?
   Ans: There's a function called `set_angle(self, angle)` declared in `servo.py` pass in 0 to see if your servo arm needs to be tuned, since the arm could be installed slightly off. If the arm is installed with an offset, you can either manually reinstall the arm (call the function so it turns to 0, then remove the arm and install it, so it faces forward) or in software just always have an offset value before passing the angle in.
6. Does the camera installation direction matter?
   Ans: No, in software you can change the orientation of the frame, so just make the camera faces forward.
7. In the naive mapping part in the report, what contents should be included in this part?
   Ans: Just explain the algorithm or the basic functionality that you implemented for "seeing" its environment.

Where to Go From Here

In this lab, you gained familiarity and comfort with doing basic IoT engineering. You learned how to work with Raspberry Pi, one of the most common platforms for IoT development. You also learned how to use sensors and actuators to interact with the environment using simple robotics.

There is certainly more to IoT than this, and I encourage you to continue your journey into the world of IoT, both in terms of gaining more extensive understanding of the concepts underlying these technologies as well as extending your breadth to other IoT technologies, which we will do in later labs (in fact, the next two labs will build upon your car to give it powerful new capabilities - we will develop networking leveraging bluetooth/wifi, as well as develop a cloud backend with machine learning). That said, the experiences you gained in this lab are general, and with them, you should now be able to attempt things like:

- Create an automatic cat feeder that dispenses food whenever it sees your cat (OpenCV already has cat detection algorithms, and you can find various sorts of motors/housings on websites like adafruit.com).
- Create an intelligent manufacturing safety system to turn off assembly machines when human yells are detected (look around at your workplace: many lighting/security/HVAC systems, robotics, electrical distribution systems, even the car you drive, are able to be monitored and controlled by interfaces where you simply adjust voltages on input wires).
- Continuously monitor health of an ill family member or friend and leverage machine learning to find problems early (regulation slows technology adoption in healthcare - many open-source projects are leveraging increasingly cheap medical sensors to do things far beyond what hospitals can do on their own).