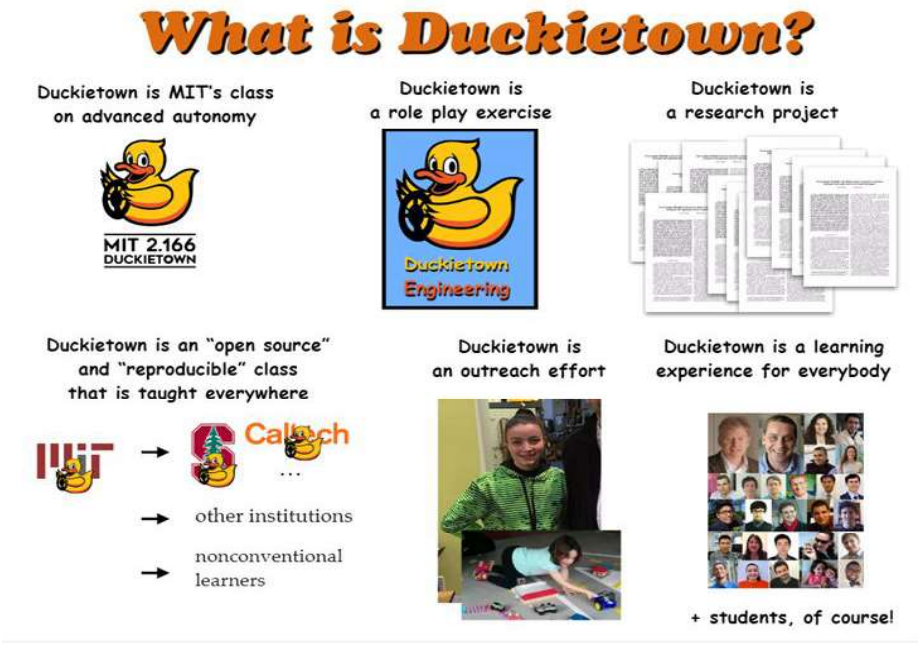
Introduction

The purpose of this project is to create a fully autonomous robot that can traverse an environment modeled after a network of roadway. The robot is based off of an open source project started by MIT. This project title is Duckietown and the robots are referred to as Duckiebots. The Duckiebot is a small wheeled robot that uses camera/vision sensor input only, wheels, microcontroller, hats, DC motors, and a battery all attached to its chassis. A fully functioning Duckiebot not only has the capability of full autonomy, but can also be controlled via remote control.

Background

Duckietown is made up of a number of things. Primarily it's part of MIT's advanced autonomy course in which undergraduate and graduate students work together for the goal of full autonomy. In addition, Duckietown is a role play exercise for the overall project and company. It is a company that the research labs created for the primary reason of helping Toyota develop a self-driving Toyota Prius, and in return MIT receives a 25 million dollar contract. The idea and goal is that this is a scalable project, therefore, the codes and algorithms they will be working on, creating, and using in the course will be practically the same ones used in a self-driving Toyota Prius. Duckietown is also an open source and reproducible class that can essentially be taught anywhere and has been picked up by other institutes such as Stanford and Caltech. Finally, Duckietown is an outreach effort that allows pre-college students interested in STEM the opportunity to familiarize themselves with autonomous control systems.

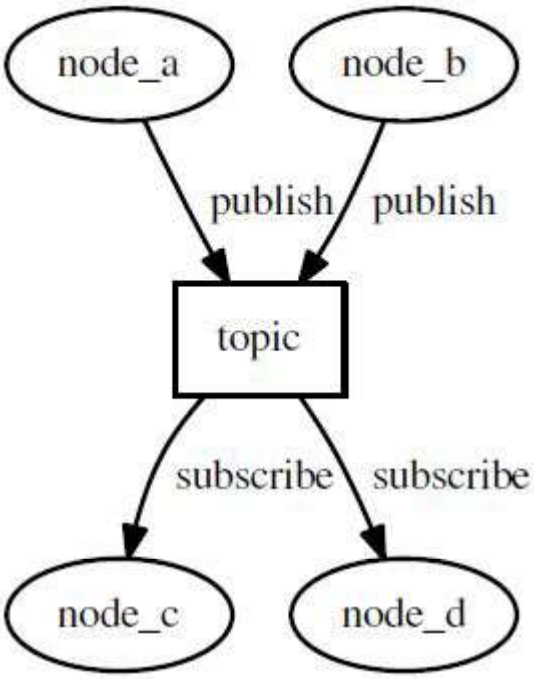


There are five levels to autonomy, level 0 being the most basic to level 4 being completely fully autonomous which means no driver assistance. Level 1 means that there is one or more specific control functions. Whereas level 2 means there are at least two primary control functions designed to work in unison to relieve the driver of control of those functions. Level 3 enables the driver to cede full control of all safety-critical functions under certain traffic or environmental conditions. Note for level 3 the driver is expected to be available for occasional control, but with sufficiently comfortable transition time. The idea for level 4 autonomy would mean the driver could safely sleep in the moving vehicle. The vehicles made by Tesla and Google are current examples of self-driving vehicles. Though Tesla's vehicles are advertised as being level 4, in reality they are level 3, due to the fact that the car still requires human guidance at times. In addition, for Google, there car relies heavily on mapping via Google maps. The problem with this is when the car is used in places where the maps are outdated like in rural areas, the mapping setup faces problems.

Software Architecture/ROS:

The softwares, software architecture and their functions in the Duckiebot computing scheme must first be understood to better integrate all hardwares and softwares in an efficient and robust autonomous system. In this project it is suggested to use a ROS as a middleware for the software architecture scheme. ROS stands for robotic operating system, but the name itself does not give the full meaning of all the functionality of what ROS does. ROS is the middleware that was chosen by MIT for the Duckietown project because it has good communications infrastructure, recording/playback of messages, remote procedure calls, distributes parameter system, a huge community, sought after skills, and is good for prototyping. To further understand what this means, a middleware is defined as, "the software that connects software components. It is the layer that lies between the operating system and applications on each side of a Distributed computer network…" The Duckiebot system works by taking an image from the camera, down samples the image to look for lane lines (middle lane line, etcetera), the controller looks for the middle of the lane, and then the controller communicates to the wheels for turning. Every middleware abstracts from hardware and hides the communication protocol. An abstraction is: not the operating system; not the application; it is right in the middle. Good middlewares have logging/playback tools and real-time analysis. For a robot there must be sensors and actuators: camera and wheels, respectively. ROS calls these 'nodes'. Node output/input is called a 'topic'. For example, the Duckiebot camera captures and generates an image, and the node/sensor generates the information/image and 'publishes' the information. Nodes that receive the information are subscribers. The actuator node 'subscribes' to the topic generated by the publisher node. In ROS nodes can be written

in different programming languages (executables), such as, Python or C++. ROS also allows for nodes written in Python to communicate with nodes written in C++. Nodes can publish multiple topics and subscribe to multiple topics. The node signals to publish a topic by communicating with the ROS master, then the master connects with the subscriber node to the publisher node.



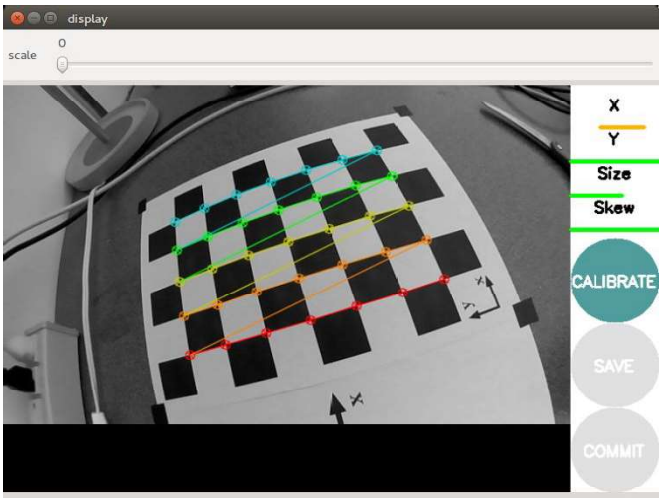
Traffic does not go through the master once connected which is through TCP/IP connection. The content and structure of a topic is defined by a message and can be thought of as an application programming interface (API). Nodes may require parameterization, such as, camera resolution, threshold, and controller gain. Within the parameter server it must initialize with the master and will not stop until the master is 'killed'. This can be set and changed at run time. XML formatting can specify almost all aspects and actions of ROS. Examples of these are the launch nodes, set/load parameters, remap topics, include other launch files, and pass commandline arguments. The use of middlewares like ROS is a very useful component of robotic architectures in integrating hardware and software together.

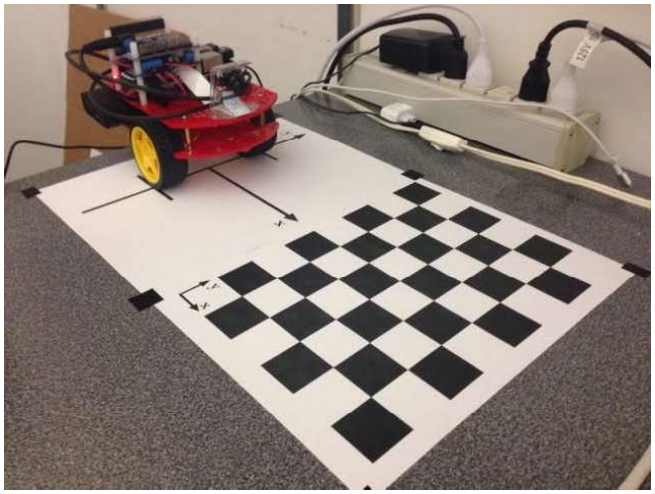
Calibration:

In any system with sensors and actuators it is good practice and essential to calibrate these components. In the Duckiebot, the sensor is the camera and the actuators are the two DC stepper motors. These must be calibrated for proper functionality. For the calibration process of the sensor (vision-based) in this project it is important to understand what the intrinsic and extrinsic parameters are and how to

extract 3D from 2D. In the scope of this project, computer vision is the only sensor input, and thus, computer vision basics must be defined by answering four questions: How to represent a pose in 3D? How are images acquired? How to estimate camera intrinsic parameters? Lastly, how to infer 3D from 2D? To answer the first, position and orientation of the robot or sensor must be determined via translation and rotation 3D vectors. By converting from typical Cartesian coordinate system to a projective geometry coordinate system known as a homogenous coordinate, scaling can easily be implemented and rotation and translation can be combined into the same framework. For the Duckiebot and world coordinate systems, this homogenous transformation can calculate the rotation and translation. A point with respect to the Duckiebot can be defined, then the world point coordinate can be calculated by multiplying the rotation and translation matrix. Moreover, for acquiring images the shape of the lens is important. By acquiring the images the perspective projection of the 3D world onto a 2D surface is obtained. The mathematical representation is the projection matrix that gives the intrinsic and extrinsic matrices with 5 and 6 DOF, respectively. The estimation and inference of 3D from 2D are then performed via the camera calibrations.

The camera calibration is the process of determining the camera's intrinsic and extrinsic parameters. In order to do this a set of correspondences between known world point coordinates and known image point coordinates in conjunction with a checkerboard are needed. For the intrinsic calibration, in theory, a minimum of two views are required, however, for robustness to noise and numerical stability, more than ten views should be used. The three parts of the algorithm MIT used for this are as follows: 1) compute the initial intrinsic parameters and ignore distortion parameters for now; 2) estimate the initial camera pose (extrinsic) with the initial intrinsic parameters; 3) run a global nonlinear optimization algorithm. Extrinsic calibration is the portion that infers 3D from 2D. To do this, the extrinsic calibration will function by estimating the extrinsic matrix. By knowing the extrinsic matrix, X can be recovered from the projection matrix. A ray in 3D can be recovered from the image point, and from the 3D ray a coordinate point on the ground can then be calculated where it hits on the ground plane. The simplest way to do this is to use planar homography which is a projective mapping from one plane to another. Mapping of points on a ground plane to the image of the camera would be an example of planar homography. In practice, extrinsic calibration would be done by laying the checkerboard (as previously mentioned) down, and lining up the Duckiebot so that it can take an image of the ground plane. The following two figures show an example image each from intrinsic and extrinsic calibrations, respectively.





The calibration of the motors are pretty straightforward and less theoretical intensive as compared to the computer vision calibration. To calibrate the stepper motors a ROS script was used to set the trim and gains. There are no encoders on these motors, therefore, the motors are running open loop voltages. Setting the trim and gain parameters is a way around the need for encoders. Trim is added until the robot can move in a straight line. After trim is configured the gains can then be optimized.

Objective - Goal of the Project

The objective for this project was to have a fully operational Duckiebot that is capable of autonomous lane following like those seen in Duckietown. However, it is determined that it is unreasonable for this due to time constraints. This is due to the fact that Duckietown at MIT is a semester long course in which students work on their Duckiebot, whereas this project which is trying to replicate the MIT Duckiebot is being done in one month. Therefore, the final objective for this project is to have a Duckiebot that is capable of remote control and has camera function.

Build Process

The following list contains the parts necessary to build a Duckiebot:

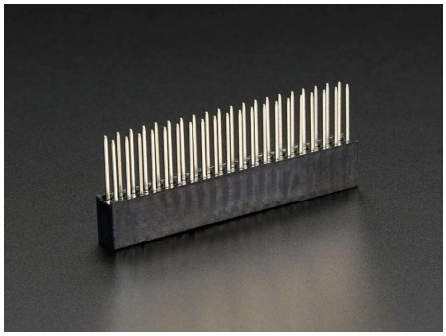
Parts List

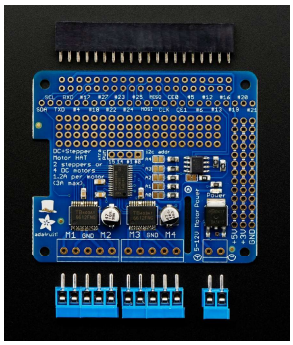
- CanaKit Raspberry Complete Starter Kit:
- Raspberry Pi 2
- USB Wifi
- HDMI cable for setup
- 8GB card (not to be used)
- Raspi case (not to be used)
- Camera mount
- Adafruit DC & Stepper Motor HAT for Raspberry Pi - Mini Kit
- Adafruit 16-Channel PWM / Servo HAT for Raspberry Pi - Mini Kit
- 2x GPIO Stacking Header for A+/B+/Pi 2
- 32 GB SD Card
- 32 GB USB disk (Sandisk)
- 1x USB to barrel cable
- Battery kit: USB
- One battery
- 2 USB A to micro USB adapters
- 1 Chassis
- 12x Plastic Standoffs for the HATS and Pi
- 1 Logitech joystick
- Fisheye camera:
- Option 1: sold as a whole:
- China: Picam fisheye camera
- Amazon: currently not available

- Option 2: buy the camera and the lens separately:
- Fisheye lens (Amazon)
- PI camera (Amazon)

Soldering

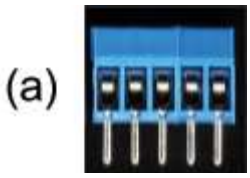
Parts: GPIO Stacking Header for A+/B+/Pi 2
Adafruit DC & Stepper Motor HAT for Raspberry Pi - Mini Kit

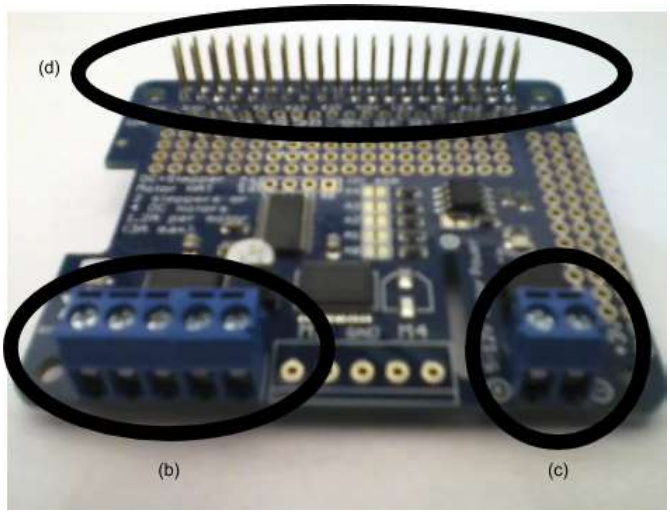




Instructions:

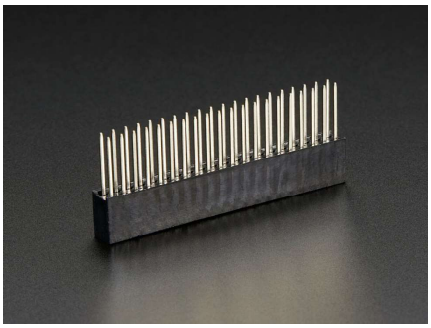
a. Make a 5 pin terminal block by sliding the included 2 pin and 3 pin terminal blocks into each other.

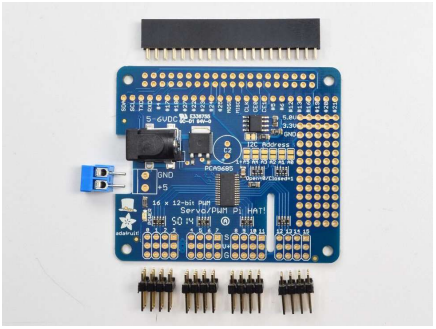




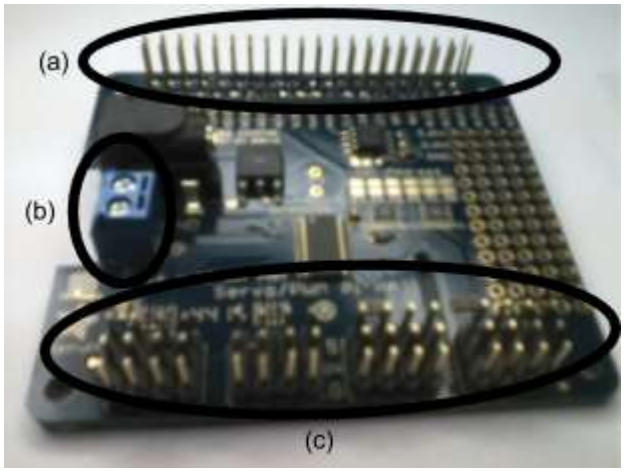
b. Slide this 5 pin block through the holes just under "M1 GND M2" on the board. Solder it on (we only use two motors and do not need connect anything at the "M3 GND M4" location)
c. Slide a 2 pin terminal block into the corner for power. Solder it on.
d. Slide in the GPIO Stacking Header onto the 2x20 grid of holes on the edge opposite opposite the terminal blocks. Solder it on

Parts: GPIO Stacking Header for A+/B+/Pi 2
Adafruit 16-Channel PWM / Servo HAT for Raspberry Pi - Mini Kit





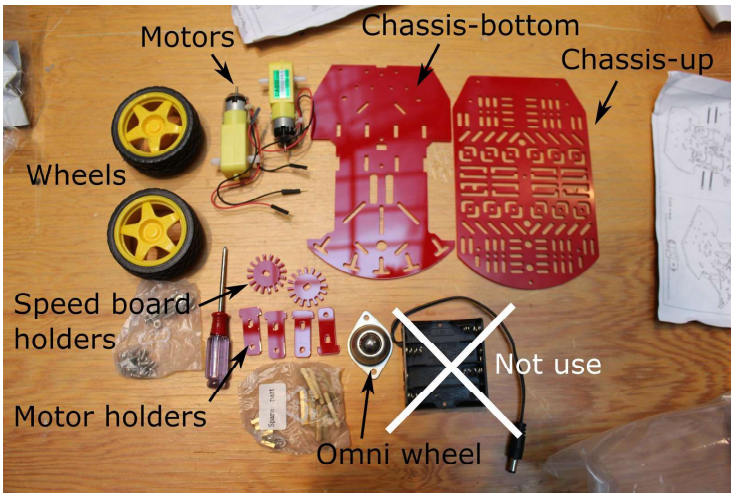
Instructions



a. Solder the GPIO Stacking Header at the top of the board, where the 2x20 grid of holes is located.
b. Solder the 2 pin terminal block next to the power cable jack
c. Solder the four 3x4 headers onto the edge of the HAT, below the words "Servo/PWM Pi HAT!"

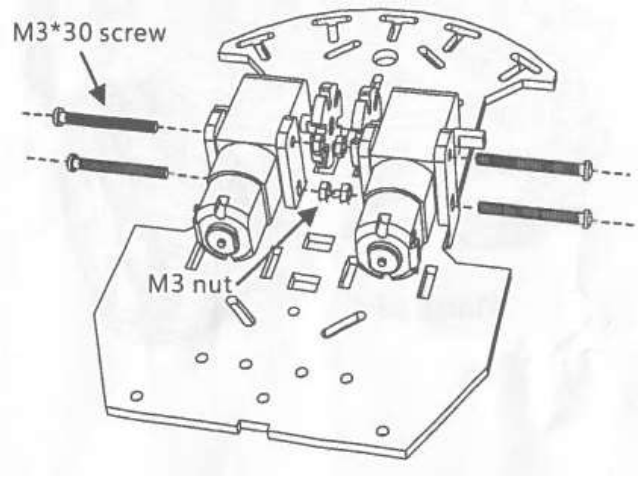
Assembly

Below are the components that come with the chassis. The battery holder and speed board holders are are not to be used.

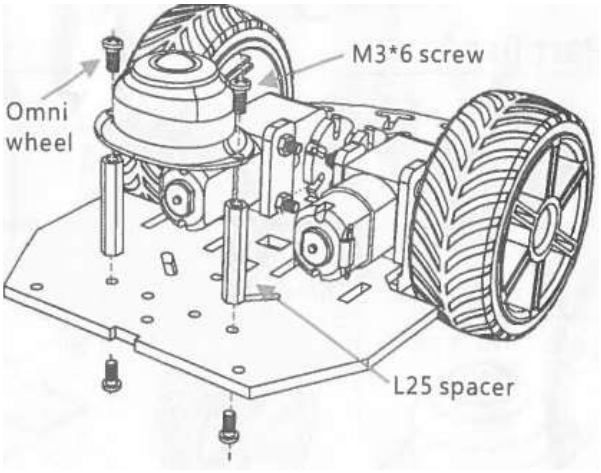


NOTICE: DO NOT MOUNT "Speed board holder"

First begin with the bottom chassis piece, two DC motors, motor holders, M3*30 screws, and M3 nuts. Assembly of these components are done as shown in the image below.

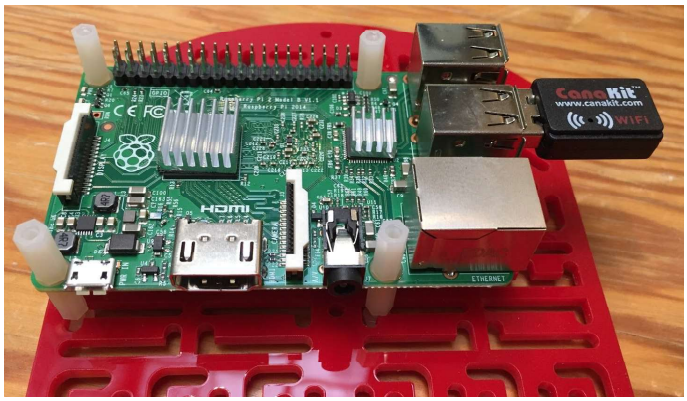


The two wheels are force fitted onto the outer motor axles. Now add the omni wheel by using two L25 spacers and four M3*6 screws.



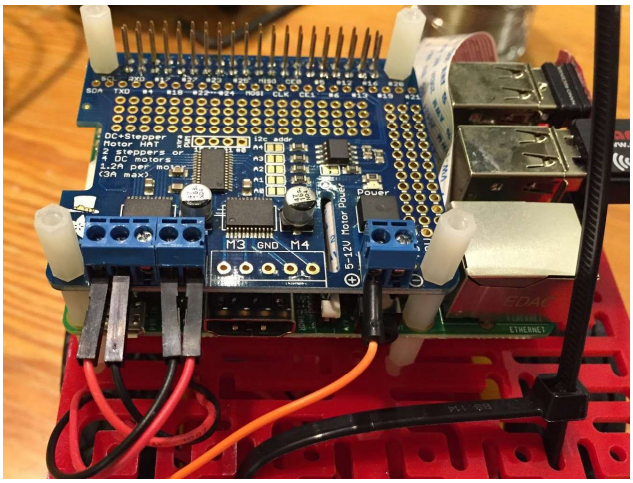
After the omni wheel has been assembled put four spacers with 4 of M3*6 screws on each corner as shown above.

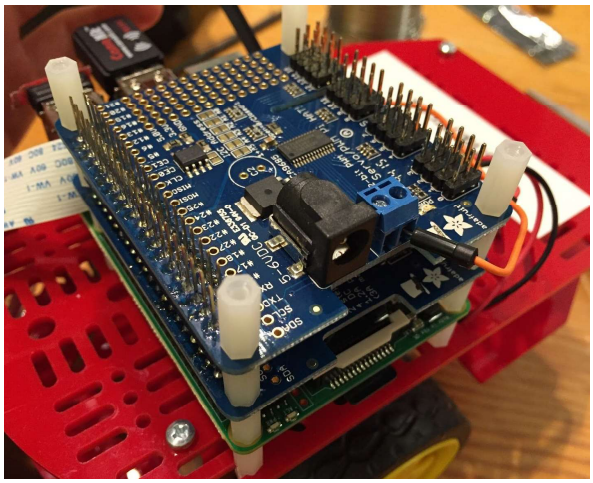
Now we can begin readying the Raspberry Pi. Take the two heatsinks and place them on the chips as shown below and insert the wifi usb adapter. Once done, take eight spacers and position them on the chassis and Pi so that they resemble the example below.



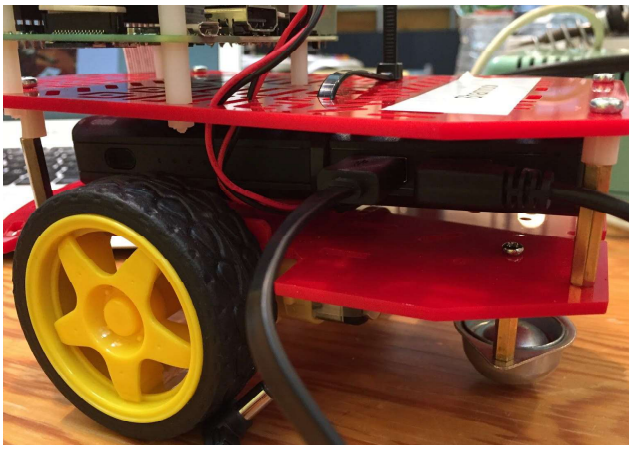
The camera and mount can be attached as shown below using M3*10 flathead screws and M3 nuts, which were for a battery holder. Remember to place the camera ribbon below the mount below screwing down. Plug in the ribbon before placing the DC & Stepper Motor HAT.

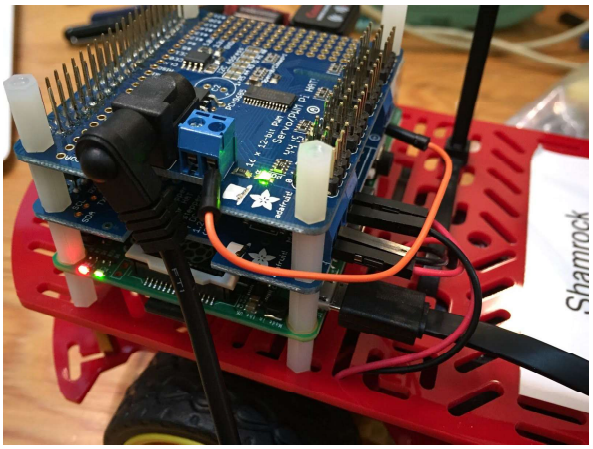
Now place the DC & Stepper Motor HAT on top of the Pi carefully pressing it down. Plug in the wires as shown below, screw them in place, add four more spacers to the corners before placing the 16-Channel PWM / Servo HAT. Place the 16-Channel PWM / Servo HAT on top again pressing down carefully and attach the jumper wire connecting the two HATs.





Finally place the battery between the two chassis levels, secure using a zip tie, and connect the two chassis levels. Connect the power cables (USB A to 5mm cable and USB cable in the battery box) to the Pi and 16-Channel PWM / Servo HAT as shown below.

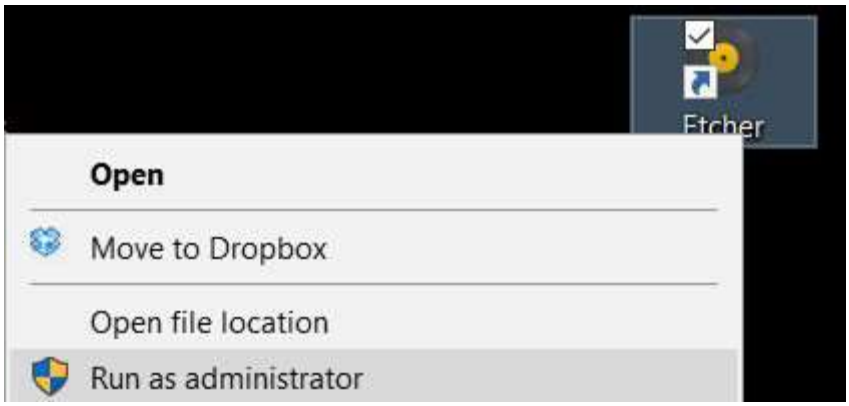




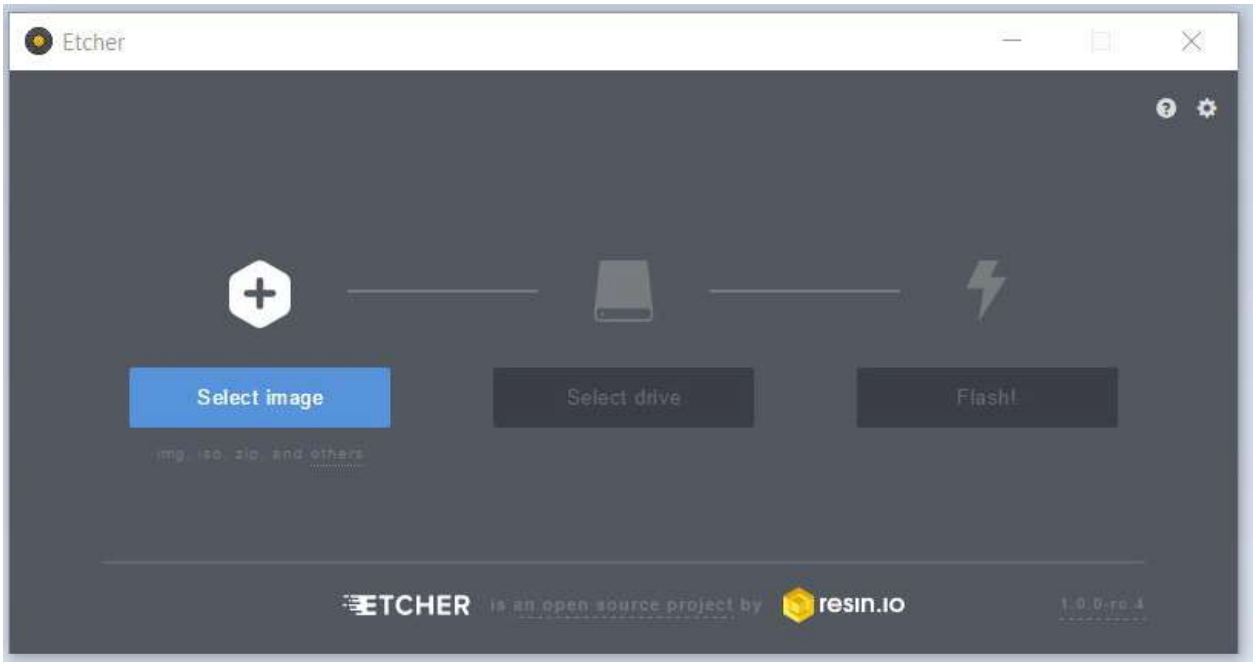
Installing Operating System

First, you must download an appropriate operating system than can run on the Raspberry PI. A highly recommend version of Linux operating system is Ubuntu MATE and at the time of the build the latest version is 16.04.02. Ubuntu MATE 16.04.02 can be downloaded from *https://ubuntu-mate.org/download/*.

In order to run an operating system on a Raspberry PI 2, first a bootable Micro SD card must be created. Download and install Etcher from *https://etcher.io/.* Etcher allows SD and USB to be turned into a bootable drive by burning the Image, ISO, and DD file. It is important to note that Etcher can do it when the ISO or similar file is in zipped folder. **Also, you must run this program as a administrator.**



Then simply select the zip file or the ISO file itself, select the memory card or USB to use as the bootable drive, and then press flash.



Note: This process may take some time. Also the file when examined using properties from in the windows explorer may show a 1GB file but the file may show up as 5GB file on etcher. This is normal. This is the new expanded size for the bootable drive.

When completed just insert the microSD card into the Raspberry PI 2 and connect a ethernet cable, USB keyboard and mouse. Then power on the PI.

Running Ubuntu and Installing Additional Packages

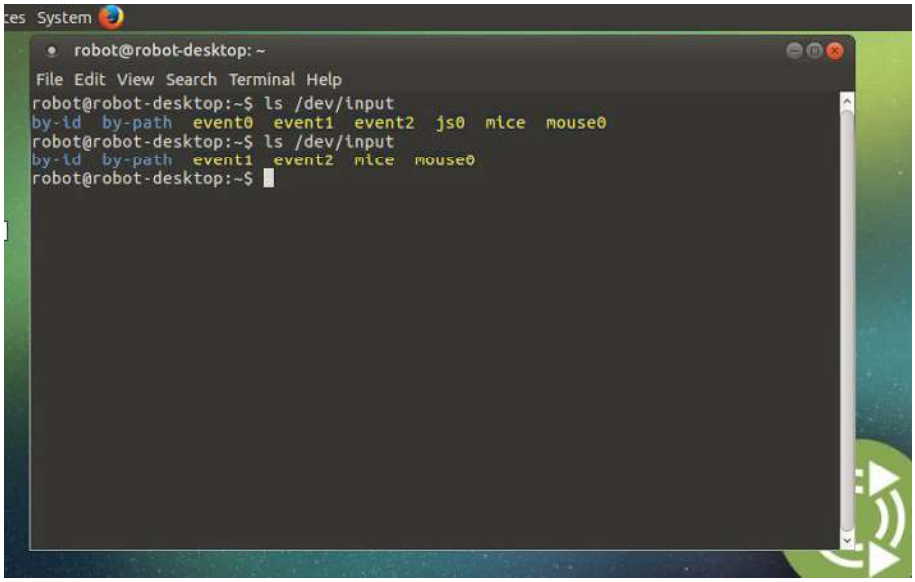
Duckiebot Credentials
Username / Password
Robot / 0000
Robot2/ 0000

Once booted up, a prompt will ask for time location, setting up wifi, languages, account credentials. The Duckiebots provided already have been set up and their username and passwords are above. After the setup is complete you will be presented with the desktop. Click on the folder labeled Robot or Robot2. In these folder(s) a folder will be labeled Duckiebot and inside will contain three main Python codes.

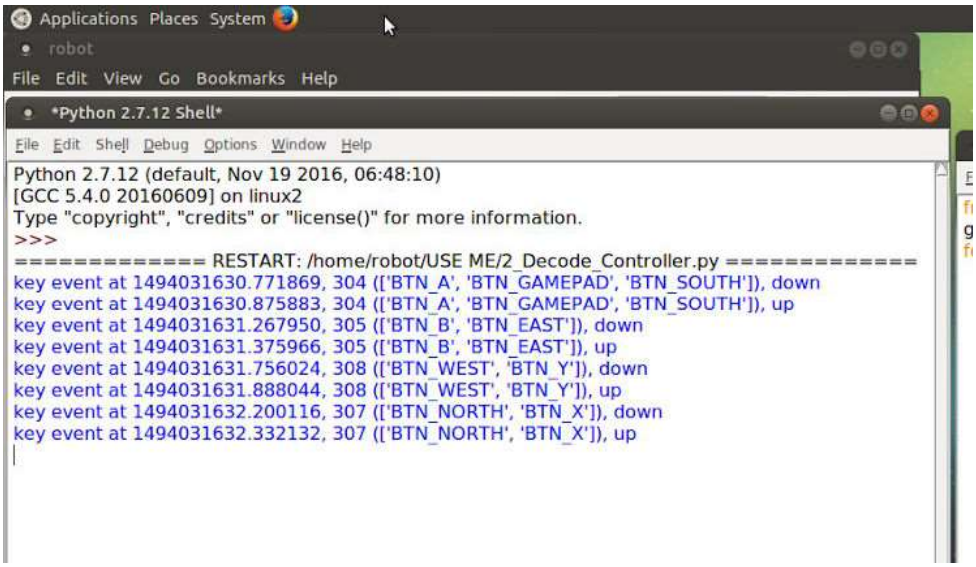
1_Find_Controller.py
2_Decode_Controller.py
3_Basic_Motor_Controls.py
NOTE: When running any of the Python scripts, be sure to use 2.6 version and not 3.5. The codes can also be found in the appendices of this report.

The first code as the name implies finds the controller. When running it properly it will identify it is either event0, event1, event2, and so on. However if this code doesn't work open the MATE Terminal (Top right corner of desktop > Applications > System Tools > MATE Terminal)

In the command line type "ls /dev/input" without the quotations. Do it twice, once with the receiver for the remote and once without the receiver for the remote. It is noted the number of eventX items are reduced. From here one can deduce that the remote is related to event2.



Once the proper event has been determined to be connected to the remote. Then running second python script will allow to be able to use the button inputs as controls. The image below runs through each of the four buttons on the right hand side of the controller.



The third code, 3_Basic_Motor_Controls incorporates the information obtained from 2_Decode_Controller to active the motors to move either forward, backwards, rotate right, and rotate left. This allows basic controls.

Necessary Packages and Configurations for Hardware

In order to use the Motor Hat, Camera, and and the remote control/gamepad multiple packages are necessary. These are installed using the pip command in the MATE Terminal and with an internet connection.

MotorHat

Installation of the Motor Hat libraries requires running the following command lines:

$$git\ clone\ https://github.com/adafruit/Adafruit-Motor-HAT-Python-Library.git$$
$$cd\ Adafruit-Motor-HAT-Python-Library$$
$$sudo\ apt-get\ install\ python-dev$$
$$sudo\ python\ setup.py\ install$$

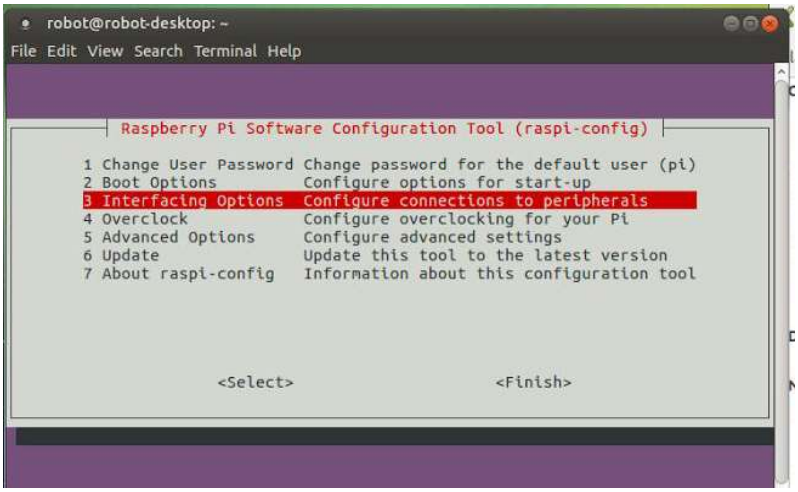
The previous lines downloads the necessary and installs the packages. Then we set the directory with respect to the Motor Hat then install python-dev and run the setup. Once this is completed the MotorHat is ready to use.

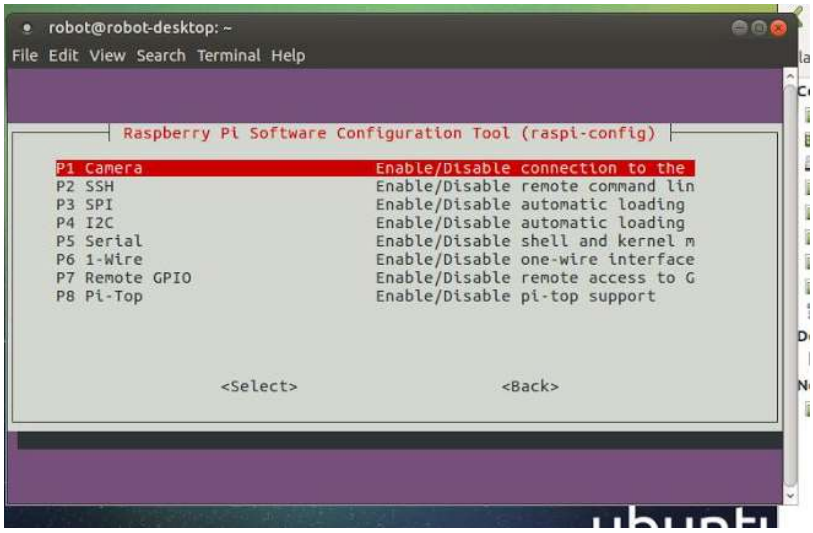
Remote Control/Gamepad

Installation of the remote control packages just requires running the following command line: *sudo pip install evdev* . The following command will download and install this python package.

Camera Packages

In order to use the camera the PI 2 needs to be configured to allow the camera to operate. This is achieved by running the command line: *sudo raspi − config* in the MATE Terminal. The terminal window will be taken over with the Raspberry Pi Software Configuration Tool (raspi-config) menu. Scroll down to *3 Interfacing Options*, see image below. Then to P1 Camera, and enable the connection. Save the changes then exit this menu. Now the camera has been enabled for use and is able to be called upon.





Results

The results of the project has produced 2 separate duckiebots that has RC control with Camera capabilities with a Ubuntu MATE interface. In addition the underlying architecture to advance forward in the development of the robot to have self autonomous driving abilities solely based on computer vision is available. As mentioned before these designs differed from MIT duckietown procedures however it can be argue this approach may lead to a better platform to build autonomous features. In addition, the detailed build document from assembly to software installation will provide a better guide to building a robot compared to MIT open source material, which allows for other students to work on.

Future Work

In the scope of this project we have reached RC control of the Duckiebot, however, there was a departure in from MIT setup and procedure in doing so. Ubuntu MATE was used and, therefore, a difference some implementation acquiring RC control as discussed in previous sections of this report. Understanding basics of Python, ROS, and Github will help with this project. Looking forward, the next steps after RC control are to follow calibration steps for the wheels/motors and camera. Calibrating requires the very first steps in implementing ROS since one of the codes to calibrate uses ROS to implement to code. After the robot can drive in a straight line and has camera image calibration, ROS will be used to take logs. MIT has provided these setup protocols and codes via Github. The first big set of steps after RC is ROS: taking a log, creating your own ROS package, and integrating the ROS package into the architecture to work with Duckiebots. Another step that can be done before ROS or after is setting up the LEDs and testing them. Things to be done from beyond this project's scope are as follows:

- Calibration of motors and camera
- LED setup and test
    - Vehicle: Headlights, Brakes, and Turn Signals
    - Simulated Environment: Traffic Lights
- Assemble Simulated Environment
- Integrating ROS: creating a package and integrating it into the Duckiebot
- Virtualbox setup
- VMware setup
- Connect Pi and Buffalo (Note: this step was not straightforward in MIT's documents)

- Getting started with simulation environment (Note: MIT is making their own simulation environment, but there are others available to consider. VREP is a robotic simulation environment used in the OSU golf cart project that might be a useful one to try, and it is relatively straightforward using with ROS)

Breaking down some of the vision-based components that will be considered after softwares and hardware are integrated properly will help in moving forward. The data processing (understanding architecture and resource constraints) and then moving to the idea of perception and belief systems could be a separate class itself. This is where the bulk of algorithm development and actual vehicle autonomy come in. Prior to this we just have a remote controlled vehicle. MIT breaks down *perception* into modules to tackle the theory and understanding behind a very complex idea which is why we still do not have fully autonomous cars on the roads now. There are many approaches into autonomy and the idea of machine learning, the following listed are the module breakdowns pertaining to perception:

- Assistive Intelligence:
    - Calibration & image rectification
    - Illumination-invariant line detection
    - Computation: raspberrry pi
- Vision-based control:
    - Approaches: Pure pursuit control, Nonlinear Bayes' filtering, Odometry calibration
    - Capabilities: Lane following, Lane departure warning
    - Scientific challenges: Constraints, Integrating perception control, Nonlinear dynamics
- Object detection:
    - Detection and localization
    - Histogram of gradient features
- Global localization:
    - Localize self
    - Markov localization
    - Monte Carlo localization
- Navigation and Planning:
    - Navigation and reactions
    - Mission planning
    - Finite state machines
    - D* and related algorithms
- SLAM:
    - Create a map of unknown environment

    - Data association and outliers
    - Incrementalization
- Safety and coordination:
    - Human and machine
    - Imperfect sensing considerations (unpredictability of humans)
    - Controlled environment sets
    - Collision avoidance
    - Trajectory adaptation

With any autonomous vehicle the idea of robustness is the goal. Robustness in perception is still a very difficult idea to implement, and it requires understanding how unpredictable an environment can be with human interaction. Developing a belief system within the algorithm scheme is essential in having the ability and robustness to predict, estimate, and optimize path planning and decision making processes. Eventually the goal will be to have a fully autonomous fleet that can safely navigate around and with other obstacles. Safety verification comes in by modeling human behavior, which is not a trivial task in itself. Taking all of this into consideration, there is a lot to do from RC control that has more theoretical complexity to it. This report is laid out to be a reference and guide in the steps to complete the ultimate goal of full autonomous Duckiebots.

References

Note: Most of the material comes straight from the MIT Duckietown slides and documentation. Source code from Eric Goebelbecker was used when switching to Ubuntu MATE and RC control references.

[1] MIT Duckietown Lectures. Massachusetts Institute of Technology, 2017, http://duckietown.mit.edu/lectures.html.

[2] Goebelbecker, E (2015). Raspberry Pi and Gamepad Programming source code. http://ericgoebelbecker.com.

Python Scripts

1_Find_Controller.py

```
## Finding GamePad Controller ---------- Section 1
from evdev import InputDevice, categorize, ecodes, KeyEvent

def find_controller():
   event0 = InputDevice('/dev/input/event0')
   event1 = InputDevice('/dev/input/event1')
   event2 = InputDevice('/dev/input/event2')
   #event3 = InputDevice('/dev/input/event3')
   controller_list = ["Logitech Gamepad F710", "Logitech Gamepad F310"]

   for controller in controller_list:
       if event0.name == controller:
           gamepad = event0
       elif event1.name == controller:
           gamepad = event1
       elif event2.name == controller:
           gamepad = event2
      # elif event3.name == controller:
       #    gamepad = event3
       else:
           print("controller not found")
           gamepad = 0
   return gamepad

gamepad = find_controller()
```

2_Decode_Controller.py

```
from evdev import InputDevice, categorize, ecodes
gamepad = InputDevice('/dev/input/event2' )
for event in gamepad.read_loop():
       if event.type == ecodes.EV_KEY:
       print(event)
```

3_Basic_Motor_Controls.py

```
#!/usr/bin/python

from evdev import InputDevice, categorize, ecodes, KeyEvent
gamepad = InputDevice('/dev/input/event2')

from Adafruit_MotorHAT import Adafruit_MotorHAT, Adafruit_DCMotor

import time
import atexit

mh = Adafruit_MotorHAT(addr=0x60)
```

```
def turnOffMotors():

   mh.getMotor(1).run(Adafruit_MotorHAT.RELEASE)
   mh.getMotor(2).run(Adafruit_MotorHAT.RELEASE)
   mh.getMotor(3).run(Adafruit_MotorHAT.RELEASE)
   mh.getMotor(4).run(Adafruit_MotorHAT.RELEASE)

atexit.register(turnOffMotors)

from picamera import PiCamera
from time import sleep

camera = PiCamera()

for event in gamepad.read_loop():
   if event.type == ecodes.EV_KEY:
       keyevent = categorize(event)
       if keyevent.keystate == KeyEvent.key_down:
           if keyevent.keycode[0] == 'BTN_A':
               print("Back")
               mh.getMotor(2).setSpeed(150)
               mh.getMotor(1).setSpeed(150)
               mh.getMotor(1).run(Adafruit_MotorHAT.FORWARD)
               mh.getMotor(2).run(Adafruit_MotorHAT.FORWARD)
               time.sleep(1.0)
               print("Release")
               mh.getMotor(1).run(Adafruit_MotorHAT.RELEASE)
               mh.getMotor(2).run(Adafruit_MotorHAT.RELEASE)

           elif keyevent.keycode[1]== 'BTN_Y':
               print("Forward")
               mh.getMotor(2).setSpeed(150)
               mh.getMotor(1).setSpeed(150)
               mh.getMotor(1).run(Adafruit_MotorHAT.BACKWARD)
               mh.getMotor(2).run(Adafruit_MotorHAT.BACKWARD)
               time.sleep(1.0)
               print("Release")
               mh.getMotor(1).run(Adafruit_MotorHAT.RELEASE)
               mh.getMotor(2).run(Adafruit_MotorHAT.RELEASE)

           elif keyevent.keycode[0]== 'BTN_B':
               print("Right")
               mh.getMotor(2).setSpeed(50)
               mh.getMotor(1).setSpeed(150)
               mh.getMotor(1).run(Adafruit_MotorHAT.BACKWARD)
               mh.getMotor(2).run(Adafruit_MotorHAT.BACKWARD)
               time.sleep(1.0)
               print("Release")
               mh.getMotor(1).run(Adafruit_MotorHAT.RELEASE)
```

```
    mh.getMotor(2).run(Adafruit_MotorHAT.RELEASE)

elif keyevent.keycode[1]== 'BTN_X':
    print("Left")
    mh.getMotor(2).setSpeed(150)
    mh.getMotor(1).setSpeed(50)
    mh.getMotor(1).run(Adafruit_MotorHAT.BACKWARD)
    mh.getMotor(2).run(Adafruit_MotorHAT.BACKWARD)
    time.sleep(1.0)
    print("Release")
    mh.getMotor(1).run(Adafruit_MotorHAT.RELEASE)
    mh.getMotor(2).run(Adafruit_MotorHAT.RELEASE)

elif keyevent.keycode == 'BTN_TR':
    print("Start Video")
    camera.start_recording('/home/robot/FinalDriveTest.h264')

elif keyevent.keycode == 'BTN_TL':
    print("Stop Video")
    camera.stop_recording()
```