Al-Azhar University, Cairo

Faculty of Engineering

Electrical Engineering Department

Electrical Self-Driving Car

B. Sc. Graduation Project

(Electronics and Communication)

Supervised by:

Dr. Usama Abdel Fattah

Presented by:

Ahmad AbdUl-Mageed

Ahmad Mohamed Awad

Mohamed Alaa Ramdan

Omar Ibrahem Elsayed Ali

July 2021

Electrical Self-Driving Car

B. Sc. Graduation Project

(Electronics and Communication)

Supervised by:

Dr. Usama Abdel Fattah

Presented by:

Ahmad AbdUl-Mageed (ahmad.abdul-mageed@outlook.com)

Ahmad Mohamed Awad (ahamedm.awadm@gmail.com)

Mohamed Alaa Ramdan (MohamedAlaa_96@azhar.edu.eg)

Omar Ibrahem Elsayed Ali (omarIbrahem839@gmail.com)

July 2021

Acknowledgment

- This work won't be done without the guidance of Allah and the support of our parents and the guidance of the senior people.

- We would like to express our immense gratitude to **Dr. Usama Abdel Fattah** for his constant support and motivation along the way that has encouraged us to come up with this project.

Abstract

- For decades self-driving cars have been a sci-fi dream and now they're becoming a reality. By 2022, it's estimated that 1 million cars with autonomous features will be on the road, **Google**; the biggest network has started working on the self-driving cars since 2010 and still developing new features and changes to give a whole new-brand level to the automated vehicles.

- And a few months before, **Tesla** have published it's auto-pilot package, can be bought from it's site for all **Tesla** cars that do not support the auto-pilot system.

- Because this project is trendy, a lot of companies wants to take a leap in this field to control the market, such as **Tesla, Uber**, **Google**, **Nivida**, and others.

- The aim and the goal for self-driving cars is that it could eliminate all road related accidents. While this would be a tremendous thing for humanity.

- The ICE-Cars

- So for that reason, we decided to make our project about **Electrical self-driving car**.

- The function of Electrical self-driving car boils down into 4 functions:

Finding Lane Line: by making the car be able to find the lane line of the road using image processing and computer vision through the camera.

Traffic-Sign detection: by making the car be able to recognize the traffic sign and make action based on that decision.

Behavioral Cloning: by making the level of driving looks like there is someone driving the car.

Advanced Lane Line Detection: by making the car recognize the curvature of the lane lines and make better decisions

List of Abbreviations

CNN	Convolutional Neural Network
AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
ADS	Automated Driving Systems
CV	Computer Vision
RPI	Raspberry pi
SSH	Secure shell
VNC	Virtual Network Computing
PID	Proportional-Integral-Derivative
PWM	Pulse Width Modulation
BIBO	Bounded Input Bounded Output
MCAL	Microcontroller Abstraction Layer
HAL	Hardware Abstraction Layer

Table of Content

Contents

Abstract 4

List of Abbreviations 6

Introduction 11

1.1 What is self-driving car? 12

1.1.2 Autonomous Car Safety and Challenges: 13

1.1.3 Levels of autonomy in self-driving cars: 13

1.2 Algorithm of Self-driving Car: 14

1.2.1 Making Decision: 14

1.3 Potential Advantages: 15

1.4 Potential Obstacles: 16

Vision and Calibration 17

2.1 Introduction 18

2.2 Distortion Correction: 19

2.3 Effects of Distortion: 21

2.4 Pinhole Camera Model and Types of Distortion: 21

2.5 Distortion Coefficients and Correction: 25

2.6 Image Formation: 27

2.7 Measuring Distortion: 27

2.8 Our Test ..... **Error! Bookmark not defined.**

2.9 The Code 30

Traffic Light Classification 33

3.1 Computer Vision 34

3.1.1 Computer Vision in Industry 35

3.2 Sensors 35

3.3 LiDAR Sensor Data 36

3.3.1 Point Clouds 37

3.3.2 Visual World 38

3.4 Image Classification Pipeline 38

3.5 Classification Techniques 40
3.6 Machine Learning 41
3.7 Separating Data 41
3.8 Training a Model 43
3.9 Images as Numerical Data 45
3.10 Color Images 47
3.10.1 Importance of Color 47
3.10.2 Pre-processing 48
3.11 Color Masking 50
3.12 Color Spaces 53
3.13 HSV and HLS 54
3.14 HSV Conversion 55
3.15 Day and Night Classification 57
3.16 Accuracy 59
3.17 Numerical labels 60
3.18 Project Overview 60
3.18.1 Traffic Light Classifier 60
3.18.2 Classification Steps 61
Distance Measurement 63
4.1 Introduction 64
4.2 Human Vision System 65
4.3 How Camera Works 67
4.4 Calculate Distance from RGB Camera 68
4.4.1 The projection on ZY Plane (Figure 5): 69
4.4.2 The projection on ZX Plane (Figure 6): 71
4.5 Measurement Code 73
Microcontroller 75
5.1 Introduction: 76
5.1.1 AVR families: 76
5.1.2 History of AVR: 77
5.2 The Atmega32 78
5.2.1 Introduction: 78
5.2.2 Specification: 79

5.2.3 Pin Descriptions: 80
The Software 82
6.1 Software Architecture 83
6.1.1 Introduction: 83
7.1.2 Microcontroller Abstraction Layer (MCAL): 84
7.1.3 Hardware Abstraction Layer (HAL): 84
7.1.4 Services: 84
7.1.5 Application 85
7.1.6 Microcontroller Layered Architecture 85
7.2 Naming Conventions 87
7.2.1 Data Types: 87
7.2.2 Local Variables: 87
7.2.3 Global Variables: 87
7.2.4 Passing Type: 87
7.2.5 Name of Variable: 87
7.2.6 Name of Function: 88
7.3 Commands 88
7.4 Flow Chart 90
7.4.1 Motor MCU Flow Chart 90
7.4.2 Main MCU Flow Chart 91
Hardware 92
8.1 Car body 93
8.2 Control Boards 94
8.3.1 Main Board Schematic 95
8.3.2 Motor Control Board 97
8.3.3 Motor Driver Board 99
8.3.4 Extension Boards 100
8.4 Sensors Used: 101
8.4.1 MPU6050 101
8.4.2 Ultrasonic Sensor HC-SR04 101
8.4.3 GPS Receiver Module (SKM53) 102
8.5 ECUs Function Block Diagram 103
Conclusion & Future Work 105

References 107
Appendix A 109

Chapter (1)

Introduction

The modern vehicle features increase not only the driver's comfort by providing automated features, but these functions are also responsible for the reduction of accidents. In common with these functions is that they depend heavily on perceived autonomous miniature vehicles need to demonstrate the reliable capability to follow their lane, to overtake vehicles blocking on their lane, to handle intersections safely according to the right-of-way traffic rule, and to park the vehicle at a sideways parking strips as fast as possible.

Though a truly driverless car is most likely still years away from being available to consumers, they are closer than many people think. Current estimates predict that by 2025 the world will see over 6 million self-driving cars on the road, and by 2035 that number will jump to almost 30 million. Trials of self-driving car services have actually-begun in some cities in the United States. And even though fully self-driving cars are not on the market yet, current technology allows vehicles to be more autonomous than ever before.

1.1 What is self-driving car?

Self-driving Cars may seem like the logical step in the evolution of personal transportation. They are conceptually simple, easy to understand and in many ways, very desirable. But they are not cars in the usual sense.

- These cars are complex systems. They are the most complex robots that humans have built.

- A self-driving car (driverless, autonomous, robotic car) is a vehicle that is capable of sensing its environment and navigating without human input.

- Self-driving cars can detect environments using a variety of techniques such as radar, GPS, computer vision, intricate systems of cameras and interconnected communication between vehicles, some models of cars now

offer features that can advanced control systems interpret sensory information to identify appropriate navigational paths, as well as obstacles and relevant signage. Self-driving cars have control systems that are capable of analyzing sensory data to distinguish between different cars on the road. This is very useful in planning a path to the desired destination.

1.1.2 Autonomous Car Safety and Challenges:

- Autonomous cars must learn to identify countless objects in the vehicle's path, from branches and litter to animals and people. Other challenges on the road are tunnels that interfere with Global Positioning Systems (GPS),

construction projects that cause lane changes, or complex decisions, like where to stop to allow emergency vehicles to pass.

- The systems need to make instantaneous decisions on when to slow down, swerve or continue acceleration normally. This is a continuing challenge for developers and there are reports of self-driving cars hesitating and swerving unnecessarily when objects are detected in or near the roadways.

1.1.3 Levels of autonomy in self-driving cars:

The U.S. National Highway Traffic Safety Administration (NHTSA) lays out six levels of automation, beginning with zero, where humans do the driving, through driver assistance technologies up to fully autonomous cars.

Here are the five levels that follow zero automation:

1. **Level 1**: Advanced driver assistance system (ADAS) aid the human driver with either steering, braking or accelerating, though not simultaneously.

ADAS includes rear-view cameras and features like a vibrating seat warning to alert drivers when they drift out of the travelling lane.

2. **Level 2**: An ADAS that can steer and either brake or accelerate simultaneously while the driver remains fully aware behind the wheel and continues to act as the driver.

3. **Level 3**: An automated driving system (ADS) can perform all driving tasks under certain circumstances, such as parking the car. In these circumstances, the human driver must be ready to re-take control and is still required to be the main driver of the vehicle.

4. **Level 4**: An ADS is able to perform all driving tasks and monitor the driving environment in certain circumstances. In those circumstances, the ADS are reliable enough that the human driver needn't pay attention.

5. **Level 5**: The vehicle's ADS act as a virtual chauffeur and do all the driving in all circumstances. The human occupants are passengers and are never expected to drive the vehicle.

1.2 Algorithm of Self-driving Car:

The self-driving is the process of capturing the images through web camera that provides machine vision capabilities to the system as well as a plethora of sensors and microcontrollers which are placed as an antenna at the top of the vehicle.

1.2.1 Making Decision:

- Fully autonomous cars can make thousands of decisions for every mile traveled. They need to do so correctly and consistently. Currently, AV designers use a few primary methods to keep their cars on the right path.

1. **Neural networks**. To identify specific scenarios and make suitable decisions, today's decision-making systems mainly employ neural networks. The complex nature of these networks can, however, make it difficult to understand the root causes or logic of certain decisions.

2. **Rule-based decision making**. Engineers come up with all possible combinations of if-then rules and then program vehicles accordingly in rule-based approaches. The significant time and effort required, as well as the probable inability to include every potential case, make this approach unfeasible.

3. **Hybrid approach**. Many experts view a hybrid approach that employs both neural networks and rule-based programming as the best solution. Developers can resolve the inherent complexity of neural networks by introducing redundancy—specific neural networks for individual processes connected by a centralized neural network. If-then rules then supplement this approach.

- The hybrid approach, especially combined with statistical-inference models, is the most popular one today.

1.3 Potential Advantages:

- being able to get things done while in traffic or on the road.
- Increase road capacity.
- Fewer traffic collisions. Experts estimate 300,000 lives can be saved Per decade.
- Higher speed limits.
- Reduction in traffic police.
- Removal of limitations on drivers - age and sobriety won't be an Issue.

1.4 Potential Obstacles:

- Liability for damage.
- Resistance by individuals to forfeit control of their cars.
- Software reliability.
- Implementation of legal framework and establishment of government regulations for self-driving cars.
- Drivers being inexperienced if situations arose requiring manual driving.
- Loss of driving-related jobs.
- Loss of privacy.

Chapter (2)

Vision and Calibration

2.1 Introduction

Robotics can essentially be broken down into a three step cycle.

- First step is to sense or perceive the world.
- The second step is to decide what to do based on that perception.
- The third step is to perform an action to carry out that decision.
- Computer Vision is a major part of the perception step in that cycle.
- 80% of building a self-driving car is perception.
- Computer Vision is the art and science of perceiving and understanding the world around you through images.
- In the case of self-driving cars, computer vision helps us detect lane markings, vehicles, pedestrians, and other elements in the environment in order to navigate safely.
- So, you might ask, "Why bother doing this at all with camera images when we have more sophisticated instruments at our disposal?"
- The answer is self-driving cars employ a suit of sophisticated sensors, but humans do the job of driving with just two eyes and one good brain.
- Now let's take a closer look why we use **cameras** instead of other **sensors** might be an advantage in developing self-driving cars.

Lidar/ Radar	Camera
Low resolution	High resolution
3D	2D
High cost	Low cost

Can we get a 3D Image from the RGB Camera?

But before starting on do anything with the pictures that gathered from camera, we need first to calibrate the camera, and make some correction to the pictures from any effect of image distortion.

2.2 Distortion Correction:

- When we talk about image distortion, we're talking about what happens when a camera looks at 3D objects in the real world and transform them into a 2D image.

- This transformation isn't perfect.

- For example, here's an image of a road and some images taken through different camera lenses that are slightly distorted.

- In these distorted images, you can see that the edges of the lanes are bent and sort of rounded or stretched outward.

- And distortion is actually changing what the shape and size of these objects appears to be.

- Because we're trying to accurately place the self-driving car in this world.

- Eventually, we'll want to look at the curve of a lane and steer the correct direction.

- But if the lane is distorted, we'll get the wrong measurement for curvature in the first place and our steering angle will be wrong

- So, the first step in analyzing camera images is to undo this distortion so that we can get correct and useful information out of them.

2.3 Effects of Distortion:

- Question: Why is it important to correct for image distortion?

- ANSWER:

1) Distortion can change the apparent size of an object in an image.

2) Distortion can change the apparent shape of an object in an image.

3) Distortion can cause an object's appearance to change depending on where it is in the field of view.

4) Distortion can make objects appear closer or farther away than they actually are.

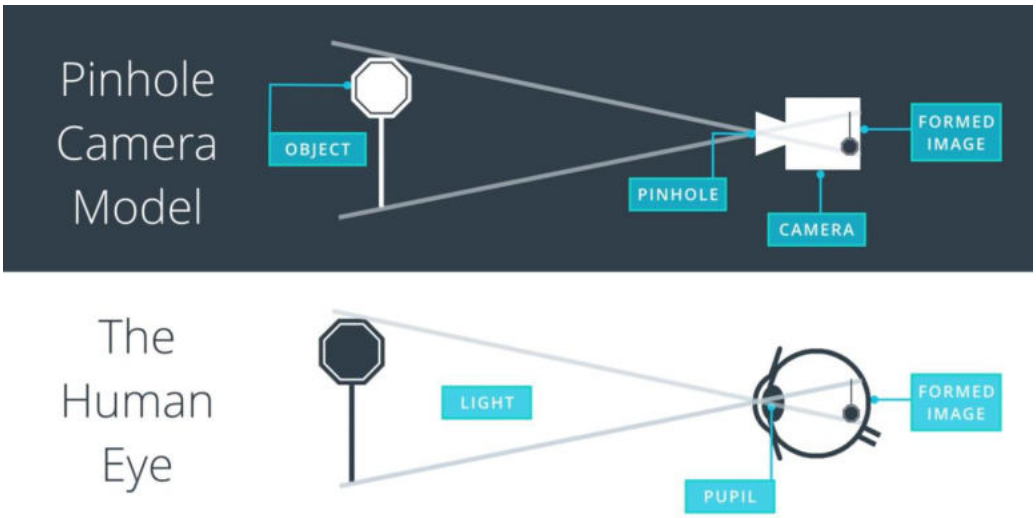
2.4 Pinhole Camera Model and Types of Distortion:

- Before we get into the code and start correcting for distortion, let's get some intuition as to how this distortion occurs.

- Here's a simple model of a camera called the pinhole camera model.

- When a camera forms an image, it's looking at the world similar to how our eyes do. By focusing the light that's reflected off of objects in the world.

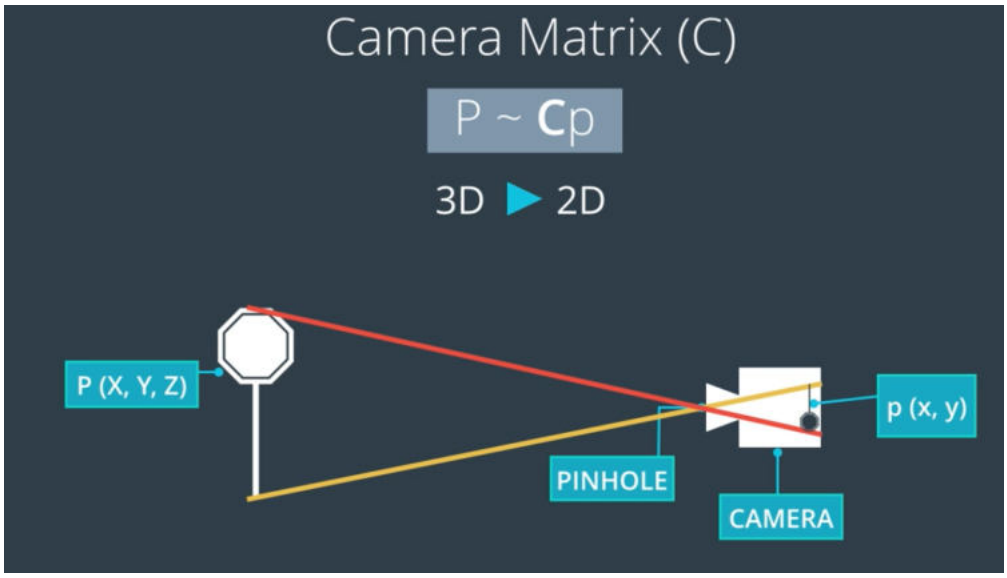
- In this case through a small pinhole, the camera focuses the light that's reflected off of a 3D traffic sign, and forms a 2D image at the back of the camera or a sensor or some film would be placed.

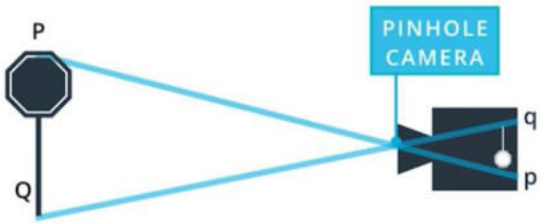


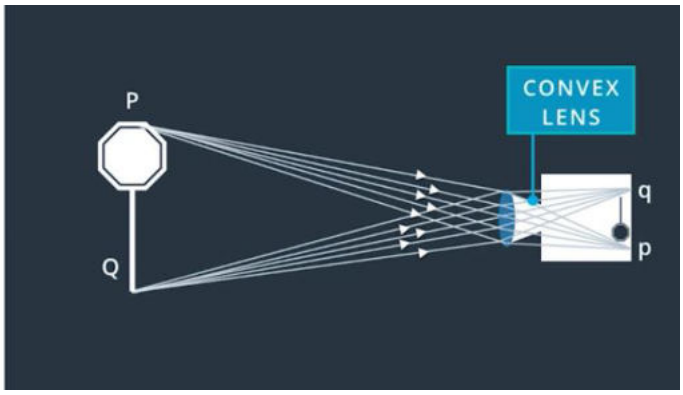
- In fact the image it forms here will be **upside down**, and **reversed** because rays of light that enter from the top of an object will continue on that angled path through the pinhole and end up at the bottom of the formed image.

- In math, this transformation from 3D object points, P of X, Y, and Z. To 2D image points, P of just X, and Y is done by a transformative matrix called **"Camera Matrix".** While we'll call C for camera and we'll need this to calibrate the camera later on.

- However, real cameras don't use tiny pinholes like this, they use **LENSES** to focus multiple light rays at a time. Which allows them to quickly form images. But lenses can introduce distortion too.







- Light rays often bend a little too much or too little at the edges of a curved lens of a camera, and this creates the effect we looked at earlier that distorts the edges of images.

- So that lines or objects appear, more or less, **curved** that they actually are. This is called **"Radial Distortion",** and it's the most common type of distortion.

- Another type, is **tangential distortion**, if the camera's lens is not aligned perfectly parallel to the imaging plane where the camera film or sensor is, this makes an image tilted. So that some objects appear further away or closer than they actually are. And this is **"Tangential Distortion".**

- There are even example of lenses that purposefully distort images like **fisheye** or **wide angle** lenses which keep radian distortion for stylistic effect.

- But for our purposes we are using this images to position our self-driving car and eventually steer it the right direction.

- So we need undistorted images that accurately reflect our real world surroundings.

– Luckily, this distortion can generally be captured by **five** numbers called **"Distortion Coefficients"**, whose values reflect the amount of **radial** and **tangential** distortion in an image.

– In severely distorted cases, sometimes even **more** than **five** coefficients are required to capture the amount of distortion.

2.5 Distortion Coefficients and Correction:

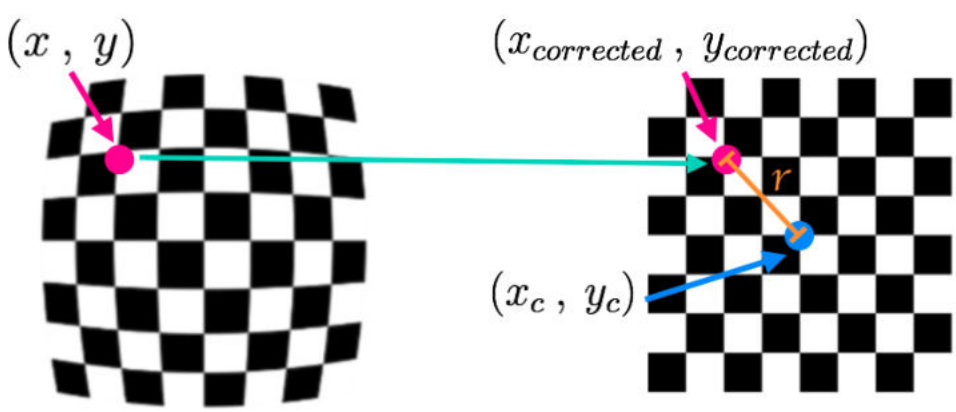
– There are 3 coefficients needed to correct for radial distortion: k1, k2, and k3.

– To correct the appearance of radially distorted points in an image, one can use a correction formula.

– In the following equations, **(x, y)** is a point in a distorted image. To undistort these points, Open-CV calculates r, which is the known distance between a point in an undistorted (corrected) image **(Corrected, Ycorrected)** and the center of the image distortion, which is often the center of that image (Xc, Yc). This center point **(Xc, Yc)** is sometimes referred to as the distortion center. These points are pictured below.

– NOTE: The distortion coefficient **k3** is required to accurately reflect **major** radial distortion (like in wide angle lenses). However, for **minor** radial distortion, which most regular camera lenses have, **k3** has a value close to or **equal** to **zero** and is **negligible**. So, in Open-CV, you can choose to ignore this coefficient this is why it appears at the end of the distortion values array: **[k1, k2, p1, p2, k3]**.

– In this course, we will use it in all calibration calculations so that our calculations apply to a **wider** variety of lenses and can correct for both **minor** and **major** radial distortion.



– There are two more coefficients that account for tangential distortion: **p1** and **p2**, and this distortion can be corrected using a different correction formula.

$$x_{corrected} = x + [2p_1xy + p_2(r^2 + 2x^2)]$$
$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2xy]$$

$$x_{distorted} = x_{ideal}(1 + k_1r^2 + k_2r^4 + k_3r^6)$$
$$y_{distorted} = y_{ideal}(1 + k_1r^2 + k_2r^4 + k_3r^6)$$

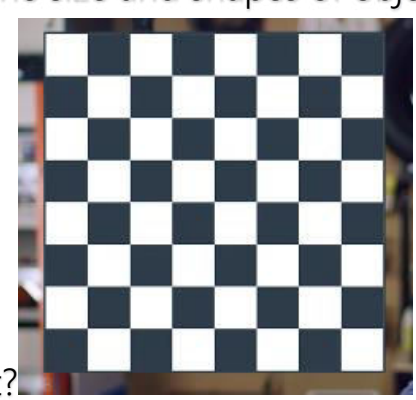
2.6 Image Formation:

– Question: What is the fundamental difference between images formed with a pinhole camera and those formed using lenses?

– Answer: Pinhole camera images are free from distortion, but lenses tend to introduce image distortion.

2.7 Measuring Distortion:

– So, we know that distortion changes the size and shapes of objects in an image, but how do we calibrate for that?



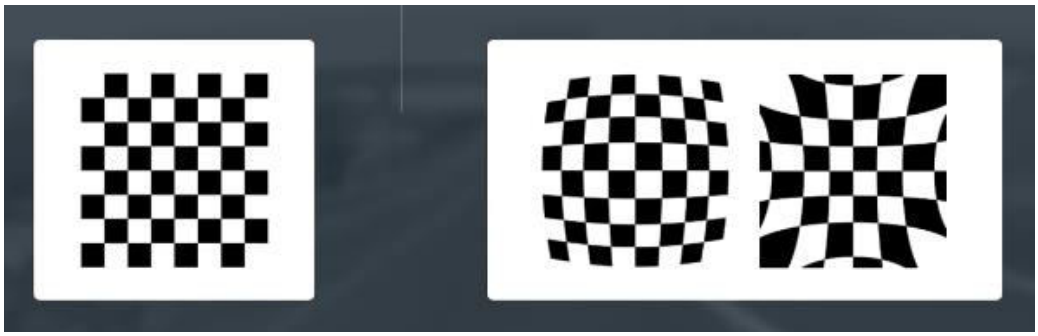
– Well, we can take pictures of **known shapes**, then we'll be able to detect and correct any distortion errors.

– We could choose any shapes to calibrate our camera, and we'll use a chessboard.

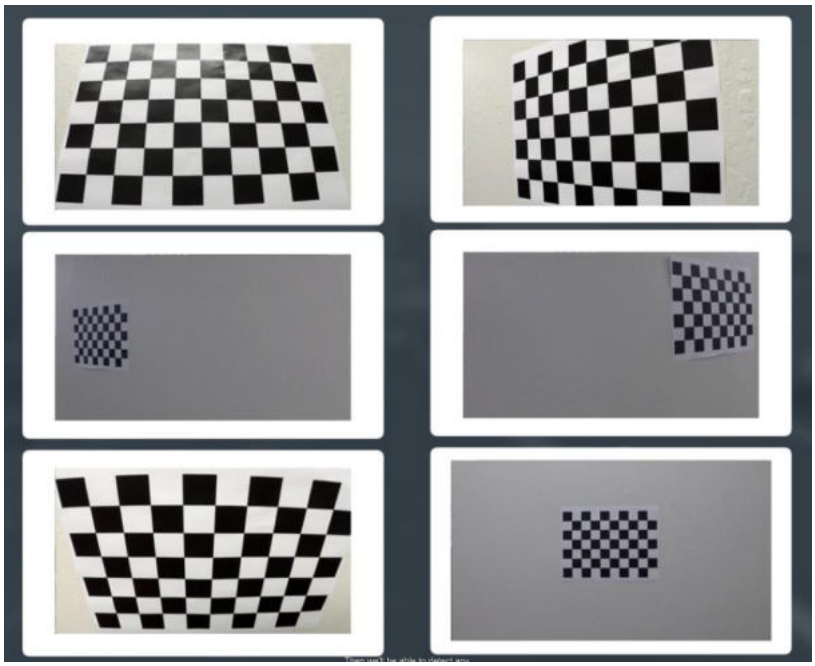
– A **chessboard** is great for calibration because its **regular high contrast pattern** makes it easy to detect automatically.

– And we know what an undistorted flat chessboard looks like.

– So, if we use our camera to take multiple pictures of a chessboard against a flat surface.

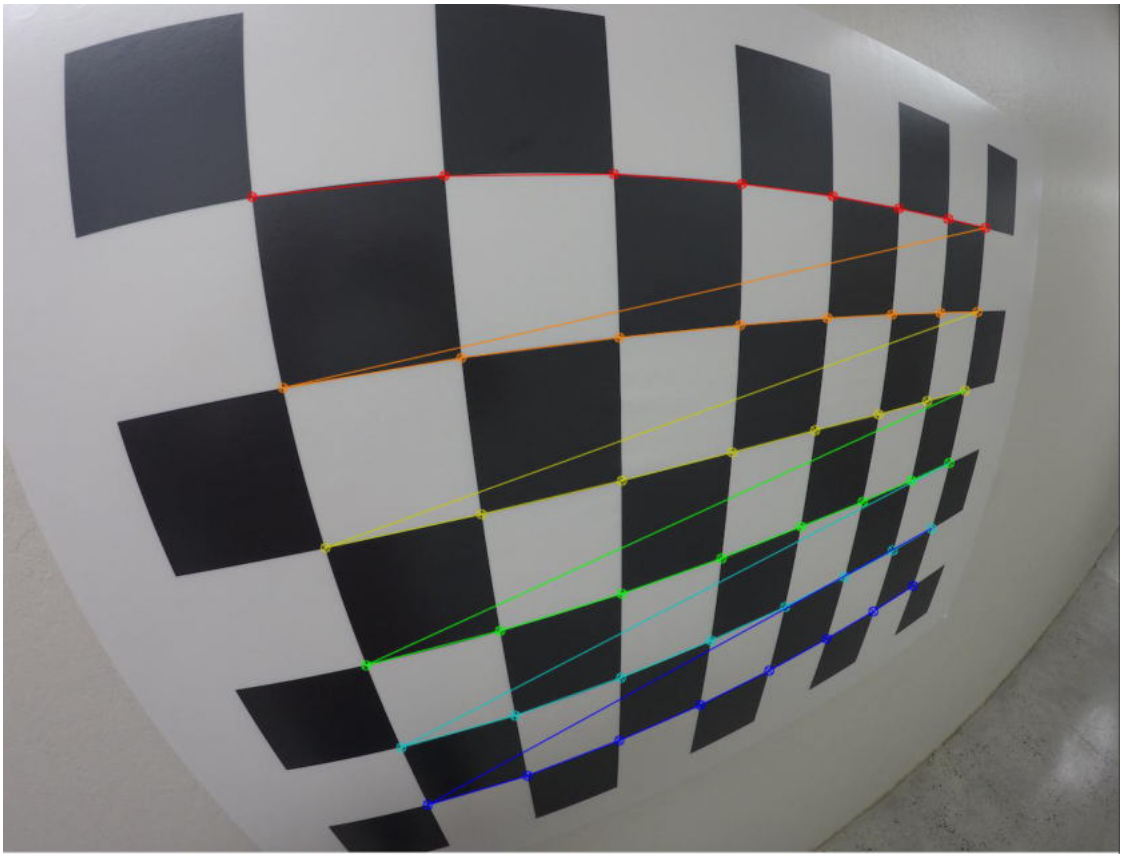


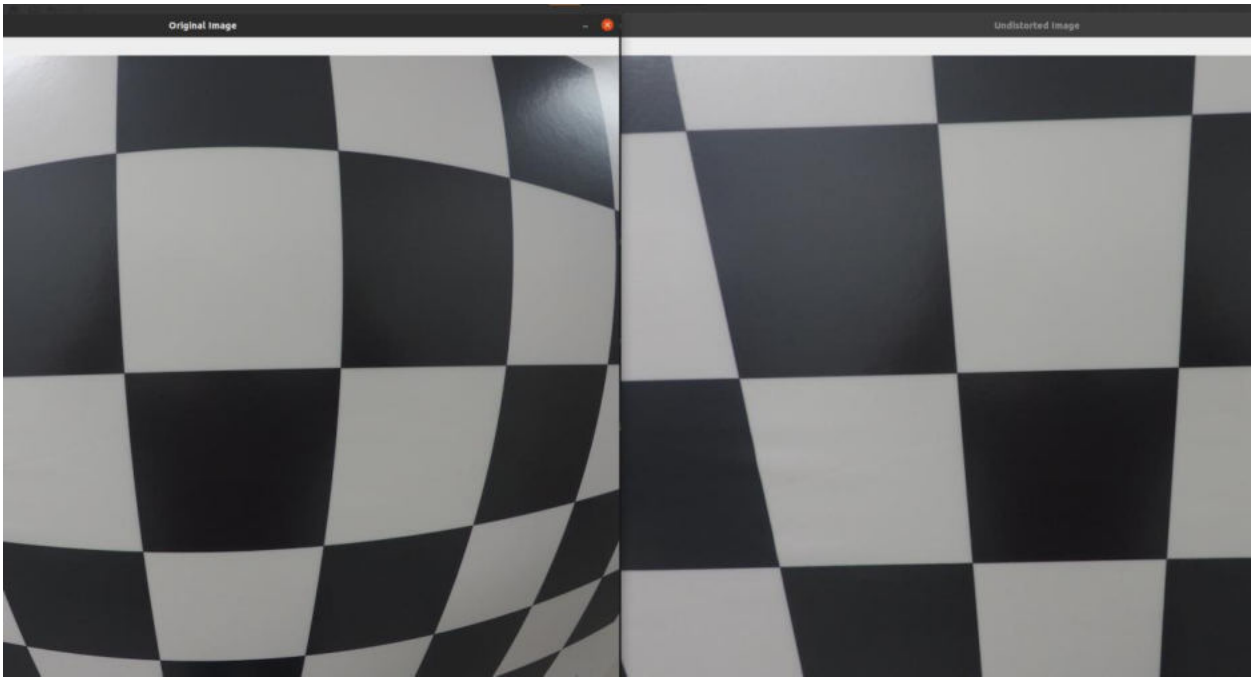
– Then we'll be able to detect any distortion by looking at the difference between the apparent size and the shape of the squares in these images, and the size and shape that they actually are.



– Then we'll use that information to calibrate our camera.

– Create a transform that maps these distorted points to undistorted points. And finally, undistort any images.




Original Image
Undistorted Image

2.8 The Code

```
# Read in the saved objpoints and imgpoints
dist_pickle = pickle.load( open( "wide_dist_pickle.p", "rb" ))
objpoints = dist_pickle["objpoints"]
imgpoints = dist_pickle["imgpoints"]

# Read in an image
img = cv2.imread('calibration_test.png')

def cal_undistort(img, objpoints, imgpoints):

    img_size = (img.shape[1], img.shape[0])

    # Do camera calibration given object points and image points
    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img_size,None,None)


    dst = cv2.undistort(img, mtx, dist, None, mtx)

    # Save the camera calibration result for later use (we won't worry about rvecs / tvecs)
    dist_pickle = {}
    dist_pickle["mtx"] = mtx
    dist_pickle["dist"] = dist
    pickle.dump( dist_pickle, open( "calibration_wide/wide_dist_pickle.p", "wb" ) )
    return dst

undistorted = cal_undistort(img, objpoints, imgpoints)

f, (ax1, ax2) = plt.subplots(1, 2, figsize=(24, 9))
f.tight_layout()
cv2.imshow('Original Image', img)
cv2.imshow('Undistorted Image', undistorted)

while(True):
    key = cv2.waitKey(0)
    if(key == ord('q')):
        quit()
```

```
        cv2.waitKey(500)

cv2.destroyAllWindows()


# Test undistortion on an image
img = cv2.imread('calibration_wide/test_image.jpg')
img_size = (img.shape[1], img.shape[0])

# Do camera calibration given object points and image points
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img_size,None,None)


dst = cv2.undistort(img, mtx, dist, None, mtx)
cv2.imwrite('calibration_wide/test_undist.jpg',dst)

# Save the camera calibration result for later use (we won't worry about rvecs / tvecs)
dist_pickle = {}
dist_pickle["mtx"] = mtx
dist_pickle["dist"] = dist
pickle.dump( dist_pickle, open( "calibration_wide/wide_dist_pickle.p", "wb" ) )
||||||
# Visualize undistortion
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
cv2.imshow('Original Image', img)
cv2.imshow('Undistorted Image', dst)

while(True):
    key = cv2.waitKey(0)
    if(key == ord('q')):
        quit()
```

```python
import numpy as np
import cv2
import glob
import matplotlib.pyplot as plt
import pickle

# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*8,3), np.float32)
objp[:,:2] = np.mgrid[0:8, 0:6].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d points in real world space
imgpoints = [] # 2d points in image plane.

# Make a list of calibration images
images = glob.glob('calibration_wide/GO*.jpg')

# Step through the list and search for chessboard corners
for idx, fname in enumerate(images):
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Find the chessboard corners
    ret, corners = cv2.findChessboardCorners(gray, (8,6), None)

    # If found, add object points, image points
    if ret == True:
        objpoints.append(objp)
        imgpoints.append(corners)

        # Draw and display the corners
        cv2.drawChessboardCorners(img, (8,6), corners, ret)
        cv2.imshow('img', img)
```

Chapter (3)

Traffic Light Classification

3.1 Computer Vision

You've been building up a solid foundation of programming and data analysis skills and, in this section, I'm excited to introduce you to one more self-driving car tool: computer vision.

Computer vision is how machines, like self-driving cars, visually perceive the world and respond to it. Imagine that you are driving a car; you have to use your senses to notice pedestrians, other cars, bicyclists, and all the road and traffic markings around you.

This is similar to how a self-driving car "sees" the world. It gathers data through cameras and other sensors, and then uses that input to safely navigate and move through the world.

In this lesson, we'll learn about common computer vision techniques that are used to analyze camera images and extract important information from them like the color or shape of different objects.

We'll also briefly talk about machine learning and how it's used in combination with computer vision to give machines a way to learn from data and recognize patterns in images.

3.1.1 Computer Vision in Industry

To help us learn about computer vision techniques and applications, we have with us an industry expert: Tarin Ziyaee, the co-founder and Chief Technology Officer of Voyage, a self-driving car company.

At Voyage, computer vision is used in a myriad of applications, for example, the recognition of what state a traffic light is in, or, the detection of lanes, and so on.

One cool thing about computer vision though, is that the techniques that you'll learn about, need not only be used with camera images, but also images created with *other sensors*. The techniques that you'll learn, will be useful for any data that has spatial coherency.

And spatially coherent data can be thought of as any data that *predictably varies over space*, like sound, for example. If you hear sound from a speaker close up it will sound very loud, but the farther you get away, the softer the sound will get. And so the volume of a sound can give you spatial information!

So, in addition to cameras, self-driving cars use sensors like radar and LiDAR, which use sound waves and lasers to gather data about a car's surroundings.

3.2 Sensors

Traditionally, RADAR is used for long-range detections, while cameras are used for rich sensory inputs.

When it comes to sensor configurations for SDCs, we have to note the details therein; LiDARs and RADARs are what are known as active sensors. That is, they sense the environment based on transmissions of energy.

Cameras on the other hand, are passive sensors; they can only sense the environment based on energy (in this case, photons) already in the scene.

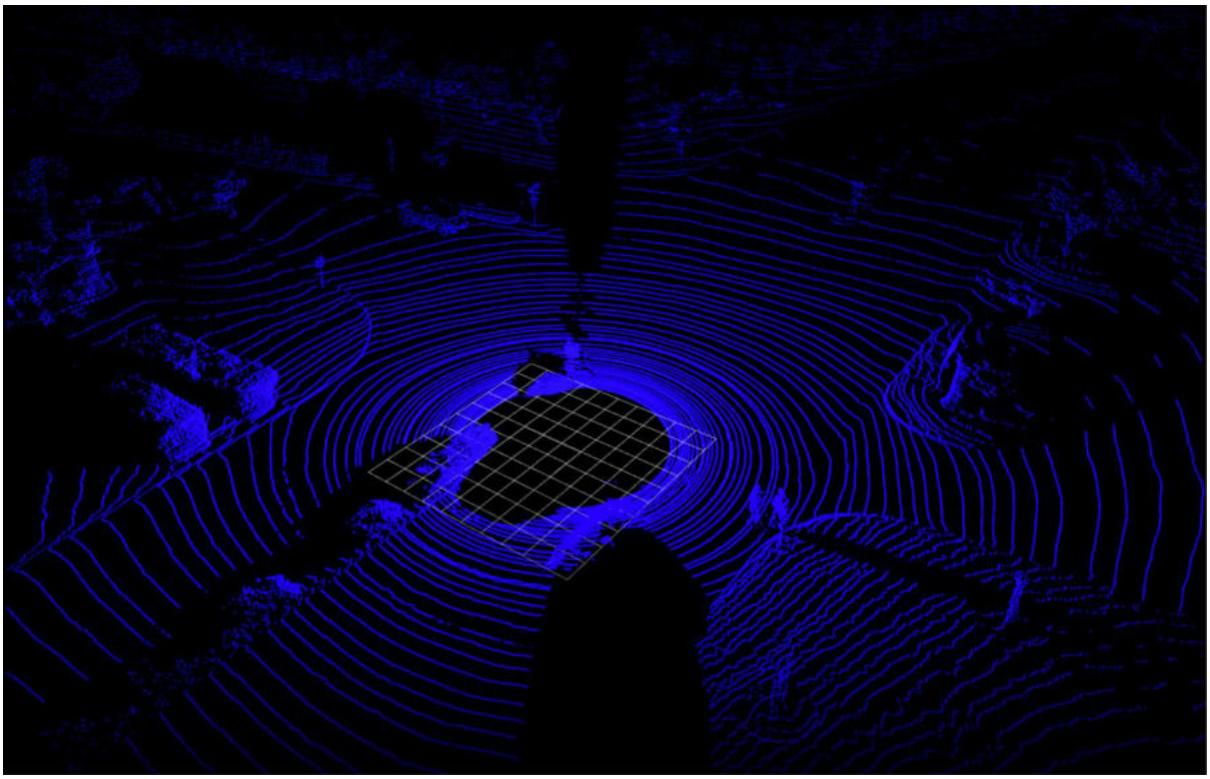
These sensor details have serious repercussions with respect to the types of algorithms we end up using to analyze this data. Computer vision has many powerful tools, but part of good design had to do not only with knowing what to do, but also what not to do when we use sensor data from multiple sources.

Now, as we mentioned, computer vision is a very powerful tool, but it should not necessarily be associated with *only* camera images. It's possible to also construct LiDAR images with a LiDAR sensor, thereby giving you measured-depth alongside well as classified pixels. Next you'll see some LiDAR sensor output and resulting data!

3.3 LiDAR Sensor Data

LiDAR stands for Light Detection and Ranging, and it is a type of sensor that uses light (a laser) to measure the distance between itself and objects that reflect light. It does this by sending a series of laser pulses out and measuring the time it takes for an object to reflect that light back to the sensor; the longer the reflection takes, the farther an object is from the sensor. In this sense, LiDAR is spatially coherent data, and can be used to create a visual world representation.

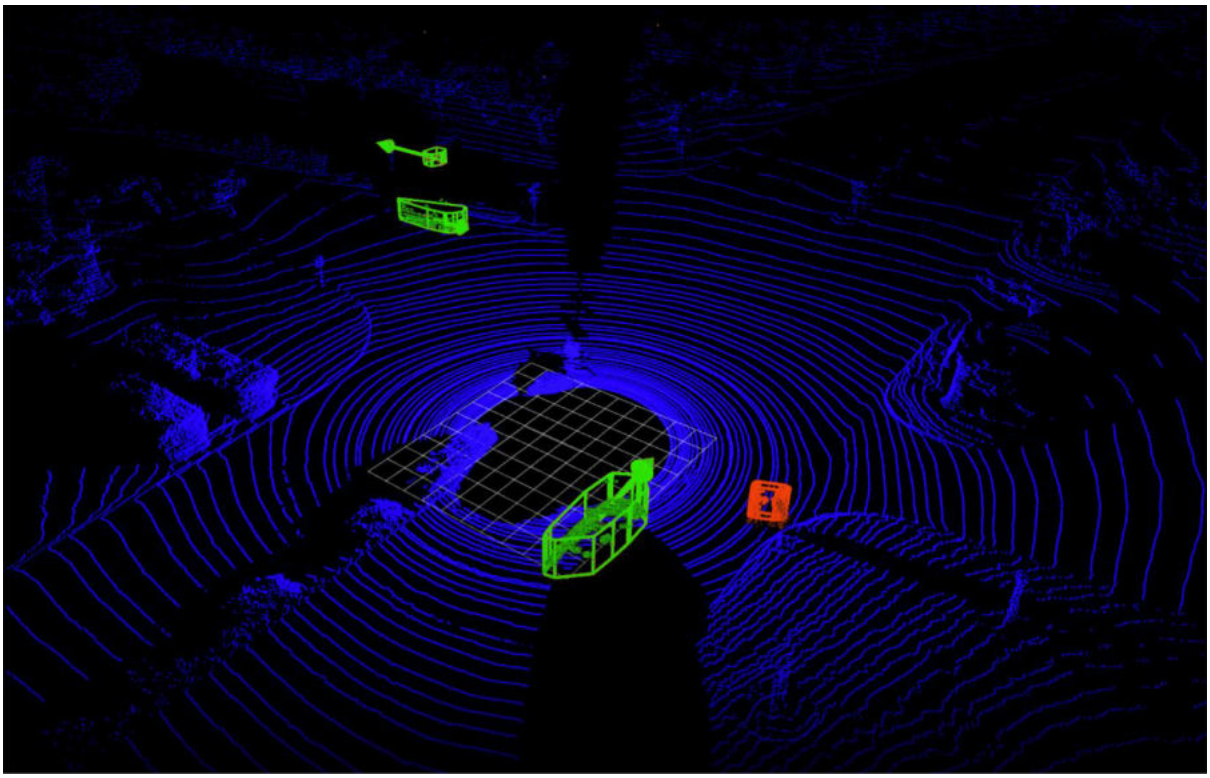
Below, you can see the output of a LiDAR that Voyage uses, it sends out pulses of light and detects the car's surroundings.



3.3.1 Point Clouds

Since LiDAR uses laser light, which sends out a thin light beam, the data it collects ends up being many single points also called point clouds. These point clouds can tell us a lot about an object like its shape and surface texture. By clustering points and analyzing them, this data provides enough information to classify an object or track a cluster of points over time!

Below, you can see the result of a classification algorithm performed on these points. The red are pedestrians and green indicates other cars.



3.3.2 Visual World

As you can see, LiDAR data provides enough spatial information to create a visual world representation, and in industry, computer vision techniques can be used to classify objects using not only camera images, but also point clouds and other types of spatially coherent data!

3.4 Image Classification Pipeline

An image classifier is an algorithm that takes in an image as input and outputs a label or "class" that identifies that image. For example, a traffic sign classifier will look at different of roads and be able to identify whether that road contains humans, cars, bikes and so on. Distinguishing and classifying each image based on its contents.

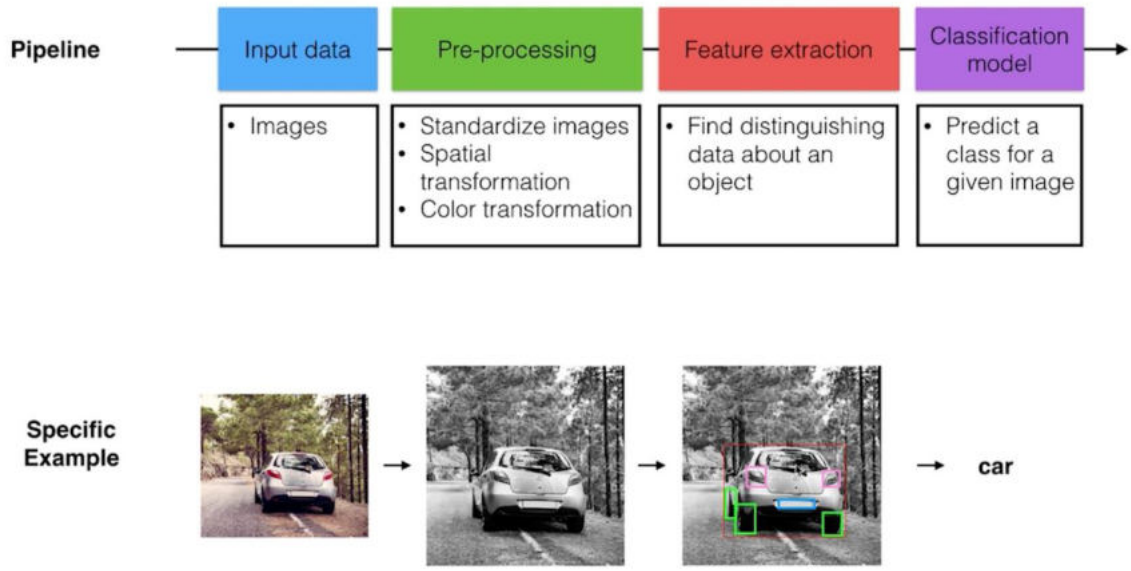
There are many types of classifiers, used to recognize specific objects or even behaviors — like whether a person is walking or running — but they all involve a similar series of steps...

First, a computer receives visual input from an imaging device like a camera. This is typically captured as an image or a sequence of images.

Each image is then sent through some pre-processing steps whose purpose is to standardize each image. Common pre-processing steps include resizing an image, or rotating it, to change its shape *or* transforming the image from one color to another - like from color to grayscale. Only by standardizing each image, for example: making them the same size, can you then compare them and further analyze them in the same way.

Next, we extract features. Features are what help us define certain objects, and they are usually information about object shape or color. For example, some features that distinguish a car from a bicycle are that a car is usually a much larger shape and that it has 4 wheels instead of two. The shape and wheels would be distinguishing features for a car. And we'll talk more about features later in this lesson.

And then, finally, these features are fed into a classification model! This step looks at any features from the previous step and predicts whether, say, this image is of a car or a pedestrian or a bike, and so on.



You'll be programming each of these classification steps manually so that you *really* understand each step.

And by the end of this lesson, you'll have all the skills you'll need to complete the final project: building a traffic light classifier, which takes in images of traffic lights and separates them into three classes: red, yellow, or green lights.

3.5 Classification Techniques

You've just seen a complete classification pipeline. For an image classifier, starting with some input images, computer vision techniques are used to process those images, and extract features, like distinguishing colors or shapes in that image.

Then, a classifier looks at these features and outputs a class, which is a label that describes the image.

A classifier should predict that images with similar shapes or colors have the same class.

We usually tell a classification model what to look for. For example, say we are looking at a bunch of images and we want to classify them into two classes: car and not-car! To classify a car, we might write a program that looks for the different parts of a car: wheels, lights, windows, and so on, and then if those things are found, we'll classify an image as a car. We decide what traits are important to look for.

3.6 Machine Learning

However, there's another way to create a classifier and that's with machine learning.

Machine learning allows a computer to figure out things on its own by giving it lots of examples. So, instead of telling a model what traits to look for, with machine learning, we'd just give it lots of images of cars and not-cars and let it learn to recognize traits that differentiate them! It can learn to recognize wheels and windows and which classification algorithm is best for *accurately* classifying any given image as car or not-car!

Now, you might be wondering: how *exactly* does a model like this learn to classify different images?

Next, we'll look at how machine learning techniques can actually be trained to classify sets of images.

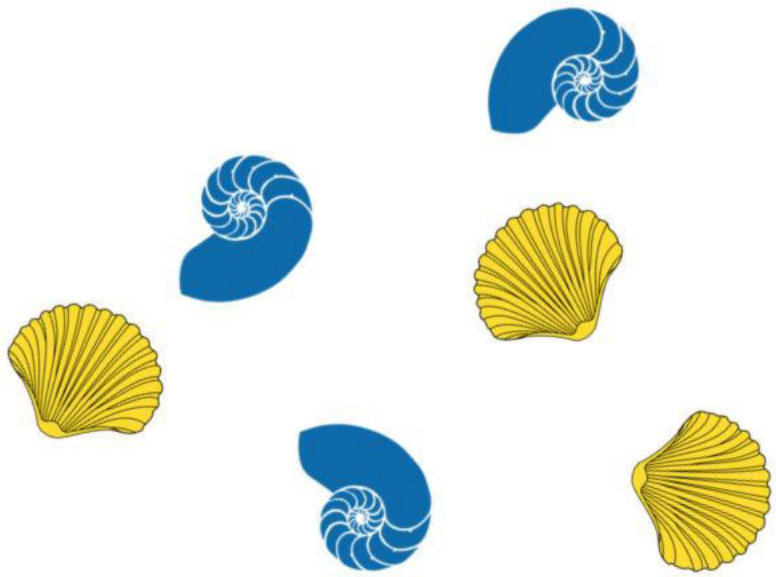
3.7 Separating Data

When we talk about machine learning, you'll often hear the terms "deep learning" and "neural networks" come up. This might conjure up an image of a brain or some strange graphic of layers of mathematical formulas and data. But at their core, all of these learning techniques are about separating data into different classes.

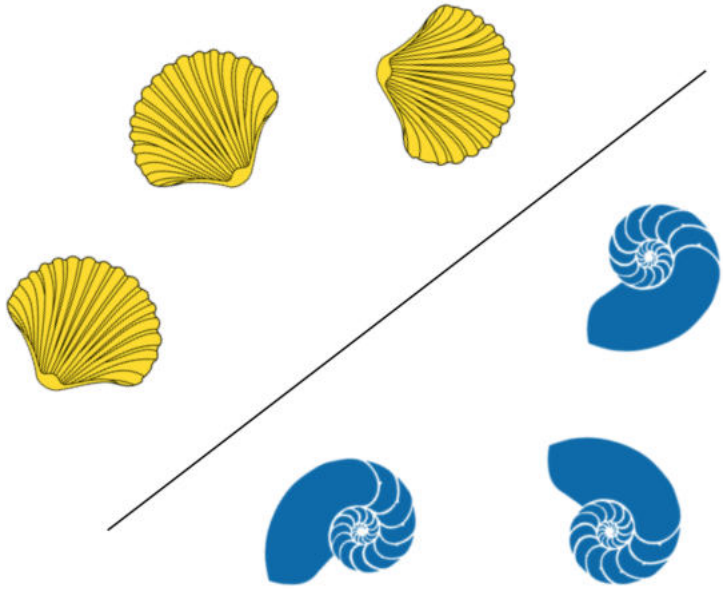
The image that this conjures in my head is of a child playing on a beach. The child sees some blue and yellow shells in the sand.

Then, someone says to the child: "put these shells into groups and draw a line between them."

Without me telling you anything else, how would you group these shells?



You might separate them by color and shape and draw a line between these two types of shells. Now, for a computer, the scenario might be that we give it individual images of shells, and just like a child, a neural network can learn how to separate these images of shells based on similarities or differences in the given examples.



After this separation step, if a network sees a *new* image, one it hasn't seen before, it sorts and classifies it based on which side of the line it falls!

Realistically, data is often a lot more complex than this, but neural networks just layer separation on top of separation layer to create more complex boundaries and group all kinds of data!

3.8 Training a Model

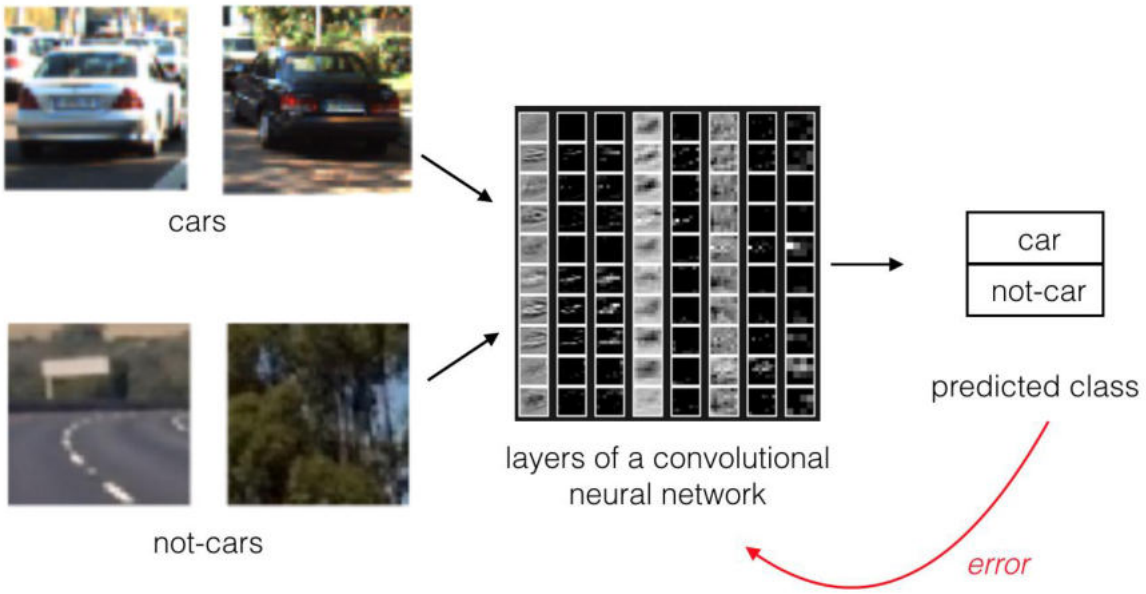
Earlier, I gave a simple example of a classification model that classifies all images as either car or not-car. And I said that using machine learning techniques, you can give a model lots of examples of cars and not-cars until it learns to recognize them. But, how exactly does this work?

Similar to how humans learn, a model has to learn by its successes and mistakes, and we often call this the training phase. At the beginning of a training phase, a classification model typically performs very badly.

The model will look at an image, try to classify it as a car or not-car and monitor the errors that it makes. For example, if a model mistakenly classifies a car as not-car, it will learn from this mistake, tweak its classification parameters and improve its performance each time it sees more images.

After many iterations, the model converges on the right set of parameters and the error rate becomes low. That's when we consider the model to be trained.

A training flow is pictured below. This is a convolutional neural network that *learns* to recognize and distinguish between images of cars and not-cars.



Now, this is a very high-level view of how to train any classification model. And the details will vary based on the type of model you use and the training algorithm you choose.

3.9 Images as Numerical Data

Every pixel in an image is just a numerical value and, we can also change these pixel values. We can multiply every single one by a scalar to change how bright the image is, we can shift each pixel value to the right, and many more operations!

Treating images as grids of numbers is the basis for many image processing techniques.

Most color and shape transformations are done just by mathematically operating on an image and changing it pixel-by-pixel.

Images as Grids of Pixels

Import resources

In [1]:
```
import numpy as np
import matplotlib.image as mpimg  # for reading in images

import matplotlib.pyplot as plt
import cv2  # computer vision library

%matplotlib inline
```

Read in and display the image

In [2]:
```
# Read in the image
image = mpimg.imread('images/waymo_car.jpg')

# Print out the image dimensions
print('Image dimensions:', image.shape)

# Change from color to grayscale
gray_image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

plt.imshow(gray_image, cmap='gray')
```

```
Image dimensions: (427, 640, 3)
```

Out[2]:
```
<matplotlib.image.AxesImage at 0x7f461b50a390>
```

In [3]:
```
# Prints specific grayscale pixel values
# What is the pixel value at x = 400 and y = 300 (on the body of the car)?

x = 400
y = 300

print(gray_image[y,x])
```

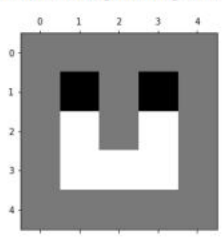
```
159
```

In [4]:
```
# Finds the maximum and minimum grayscale values in this image

max_val = np.amax(gray_image)
min_val = np.amin(gray_image)

print('Max: ', max_val)
print('Min: ', min_val)
```

```
Max:  255
Min:  2
```

In [6]:
```
# Create a 5x5 image using just grayscale, numerical values
tiny_image = np.array([[120, 120, 120, 120, 120],
                       [120, 0, 120, 0, 120],
                       [120, 250, 120, 250, 120],
                       [120, 250, 250, 250, 120],
                       [120, 120, 120, 120, 120]])

# To show the pixel grid, use matshow
plt.matshow(tiny_image, cmap='gray')

## TODO: See if you can draw a tiny smiley face or something else!
## You can change the values in the array above to do this
```

Out[6]:
```
<matplotlib.image.AxesImage at 0x7f45e23632b0>
```



In [ ]:

3.10 Color Images

Color images are interpreted as 3D cubes of values with width, height, and depth!

The depth is the number of colors. Most color images can be represented by combinations of only 3 colors: red, green, and blue values; these are known as RGB images. And for RGB images, the depth is 3!

It's helpful to think of the depth as three stacked, 2D color layers. One layer is Red, one Green, and one Blue. Together they create a complete color image.

3.10.1 Importance of Color

In general, when you think of a classification challenge, like identifying lane lines or cars or people, you can decide whether color information and color images are useful by thinking about your own vision.

If the identification problem is easier in color for us humans, it's likely easier for an algorithm to see color images too!

RGB colorspace

Import resources

In [1]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

%matplotlib inline
```

Read in an image ¶

In [2]:
```
# Read in the image
image = mpimg.imread('images/wa_state_highway.jpg')

plt.imshow(image)
```

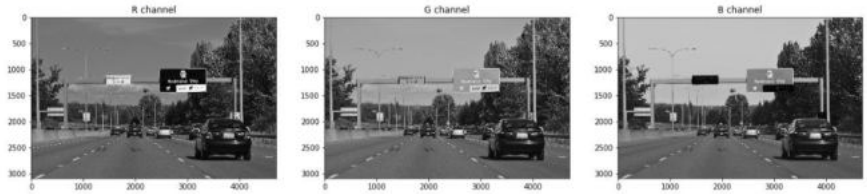
Out[2]:
```
<matplotlib.image.AxesImage at 0x7f262c47bd68>
```

RGB channels

Visualize the levels of each color channel. Pay close attention to the traffic signs!

In [3]:
```
# Isolate RGB channels
r = image[:,:,0]
g = image[:,:,1]
b = image[:,:,2]

# Visualize the individual color channels
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,10))
ax1.set_title('R channel')
ax1.imshow(r, cmap='gray')
ax2.set_title('G channel')
ax2.imshow(g, cmap='gray')
ax3.set_title('B channel')
ax3.imshow(b, cmap='gray')

## Which area has the lowest value for red? What about for blue?
```

Out[3]:
```
<matplotlib.image.AxesImage at 0x7f262b397b38>
```



3.10.2 Pre-processing

Pre-processing images is all about standardizing input images so that you can move further along the pipeline and analyze images in the same way.

Really common pre-processing steps include:

Changing how an image looks spatially, by using geometric transforms which can scale an image, rotate it, or even change how far away an object appears, and

Changing color schemes, like choosing to use grayscale images over color images.

Cropping and Resizing

Import resources

In [1]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

import numpy as np
import cv2

%matplotlib inline
```

Read in the first image of a stop sign

In [9]:
```
# Read in the image
walk = mpimg.imread('images/walk.jpg')

print('Image shape: ', walk.shape)
plt.imshow(walk)
```

```
Image shape:  (168, 300, 3)
```

Out[9]:
```
<matplotlib.image.AxesImage at 0x7f402e6102b0>
```

Read in the second image

In [10]:
```
# Read in the image
yield1 = mpimg.imread('images/yield.jpg')

print('Image shape: ', yield1.shape)
plt.imshow(yield1)
```

```
Image shape:  (604, 1024, 3)
```

Out[10]:
```
<matplotlib.image.AxesImage at 0x7f402e5f05c0>
```

Crop this image so that it resembles the first image

In [12]:
```
# To crop and image, you can use image slicing
# which is just slicing off a portion of the image array

# Make a copy of the image to manipulate
image_crop = np.copy(yield1)

# Define how many pixels to slice off the sides of the original image
row_crop = 98
col_crop = 250

# Using image slicing, subtract the row_crop from top/bottom and col_crop from left/right
image_crop = yield1[row_crop:-row_crop, col_crop:-col_crop, :]

plt.imshow(image_crop)
```

Out[12]:
```
<matplotlib.image.AxesImage at 0x7f402e525a58>
```

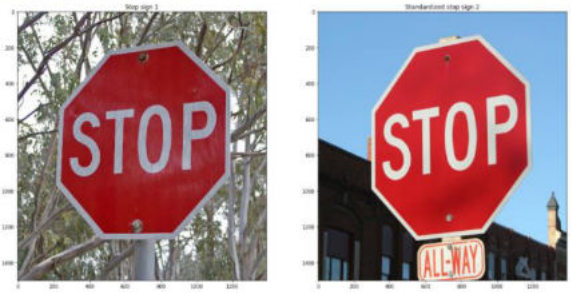
Resize the cropped image to be the same as the first

Recall that the shape of the first image is `(1500, 1389, 3)`

In [5]:
```
# Use OpenCV's resize function
standardized_im = cv2.resize(image_crop, (1389, 1500))

print('Image shape: ', standardized_im.shape)

# Plot the two images side by side
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
ax1.set_title('Stop sign 1')
ax1.imshow(stop1)
ax2.set_title('Standardized stop sign 2')
ax2.imshow(standardized_im)
```

```
Image shape:  (1500, 1389, 3)
```

Out[5]:
```
<matplotlib.image.AxesImage at 0x7f402e6d1a58>
```



Compare these images

Now you should be able to compare these images pixel by pixel! We'll add up the red channel values in each image, and they should be fairly close, which means we can use this similarity to characterize these images.

For comparison, we'll also show what happens when you perform this comparison using the original `stop_sign2.jpg`.

In [ ]:
```
# Sum all the red channel values and compare
red_sum1 = np.sum(stop1[:,:,0])
red_sum2 = np.sum(standardized_im[:,:,0])

print('Sum of all red pixel values in the first stop sign image: ', red_sum1)
print('Sum of red pixel values in the second, standardized image: ', red_sum2)

red_sum_orig = np.sum(stop0[:,:,0])

print('\nFor comparison, the sum of red pixels in the non-standardized image: ', red_sum_orig)
```

In [ ]:
```
## Note: you have been given two other images:
# 'yield.jpg' and 'walk.jpg'
# You can look at these images and see what kind of RGB values might distinguish them
```

3.11 Color Masking

We know about how color images are represented numerically, and next, I want to show you how color can be used in image analysis and transformation. We'll start by learning how to use information about the colors in an image to isolate a particular area in an image. This is easiest to understand in an example.

We'll be selecting an area of interest using a color threshold. Color thresholds are used in a number of applications including extensively in computer graphics and video; a common use is with a green screen.

A green screen is used to layer two images or video streams based on identifying and replacing a large green area.

So, how does all this work?

Well, the first step is to isolate the green background, and then replace that green area with an image of your choosing. Next, let's see how to do this programmatically with the image shown above!

Color Masking, Green Screen

Import resources

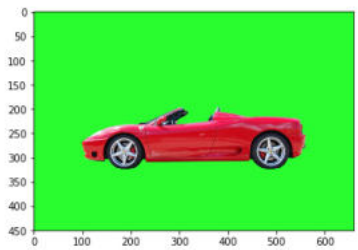
In [7]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

import numpy as np
import cv2

%matplotlib inline
```

Read in and display the image

In [8]:
```
# Read in the image
image = mpimg.imread('images/car_green_screen.jpg')

# Print out the image dimensions (height, width, and depth (color))
print('Image dimensions:', image.shape)

# Display the image
plt.imshow(image)
```

```
Image dimensions: (450, 660, 3)
```

Out[8]:
```
<matplotlib.image.AxesImage at 0x7f1322f41940>
```



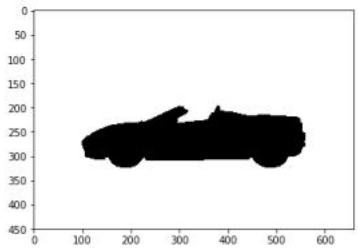
Define the color threshold

In [9]:
```
# Define our color selection boundaries in RGB values
lower_green = np.array([0,180,0])
upper_green = np.array([100,255,100])
```

Create a mask

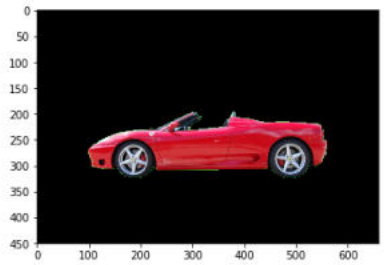
In [21]:
```
# Define the masked area
mask = cv2.inRange(image, lower_green, upper_green)

# Vizualize the mask
plt.imshow(mask, cmap='gray')
#plt.imshow(mask)
```

Out[21]:
```
<matplotlib.image.AxesImage at 0x7f1321257320>
```



In [11]:
```
# Mask the image to let the car show through
masked_image = np.copy(image)

masked_image[mask != 0] = [0, 0, 0]

# Display it!
plt.imshow(masked_image)
```

Out[11]: <matplotlib.image.AxesImage at 0x7f1322e8f860>

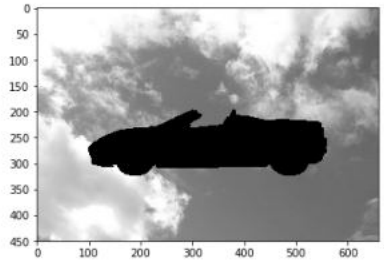


TODO: Mask and add a background image

In [41]:
```
# Load in a background image, and convert it to RGB
b = mpimg.imread('images/sky.jpg')
print(b.shape)
plt.imshow(b)
#g = background_image[:,:,1]
## TODO: Crop it or resize the background to be the right size (450x660)
# Hint: Make sure the dimensions are in the correct order!
bg = b[0:-125, 0:-364]
plt.imshow(bg)
#bg[mask != 0] = [0, 0, 0]

bg = cv2.cvtColor(bg, cv2.COLOR_RGB2GRAY)
bg[mask == 0] = 0

## TODO: Mask the cropped background so that the car area is blocked
# Hint: mask the opposite area of the previous image
#lower = np.array([100, 0, 100])
#high = np.array([255, 180, 255])
#mask = cv2.inRange(bg_image_crop, lower, high)
plt.imshow(bg)
b = cv2.cvtColor(bg, cv2.COLOR_GRAY2RGB)
plt.imshow(b)
## TODO: Display the background and make sure
```

```
(575, 1024, 3)
```

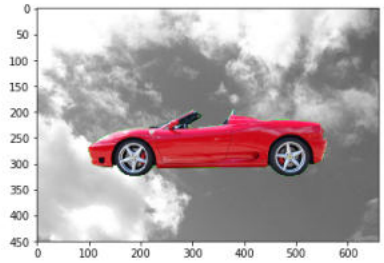
Out[41]: <matplotlib.image.AxesImage at 0x7f1320cdf588>



TODO: Create a complete image

In [36]:
```
## TODO: Add the two images together to create a complete image!
c = masked_image + b
plt.imshow(c)
```

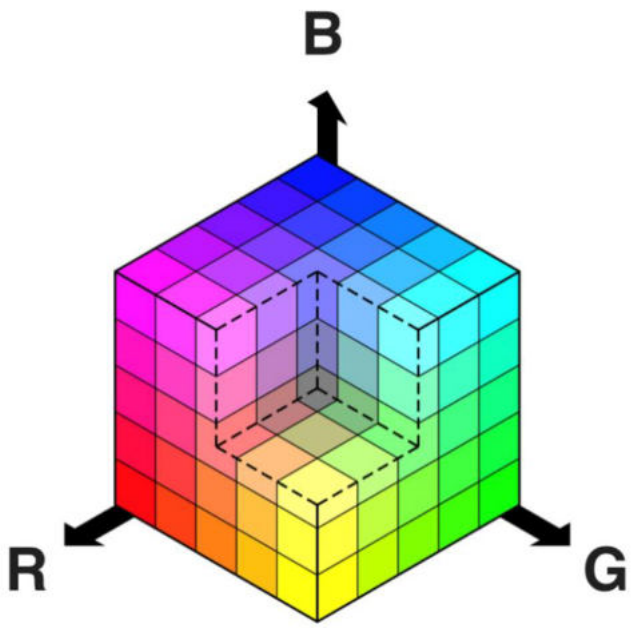
Out[36]: <matplotlib.image.AxesImage at 0x7f132112f3c8>



3.12 Color Spaces

A color space is a specific organization of colors; color spaces provide a way to categorize colors and represent them in digital images.

RGB is red-green-blue color space. You can think of this as a 3D space, in this case a cube, where any color can be represented by a 3D coordinate of R, G, and B values. For example, white has the coordinate (255, 255, 255), which has the maximum value for red, green, and blue.



There are many other ways to represent the colors in an image besides just composed of red, green, and blue values.

There is also **HSV** color space (hue, saturation, and value), and **HLS** space (hue, lightness, and saturation). These are some of the most commonly used color spaces in image analysis.

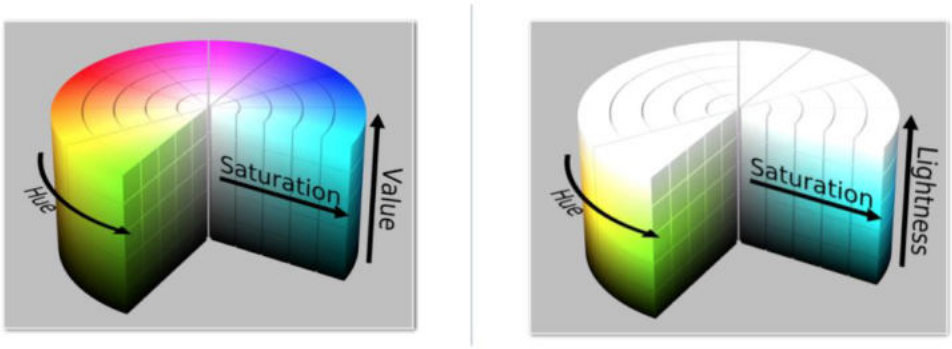
To get some intuition about these color spaces, you can generally think of **Hue** as the value that represents color independent of any change in brightness. So if you imagine a basic red paint color, then add some white

to it or some black to make that color lighter or darker -- the underlying color remains the same and the hue for all of these colors will be the same.

3.13 HSV and HLS

On the other hand, Lightness and Value represent different ways to measure the relative lightness or darkness of a color. For example, a dark red will have a similar hue but much lower value for lightness than a light red. Saturation also plays a part in this; saturation is a measurement of colorfulness. So, as colors get lighter and closer to white, they have a lower saturation value, whereas colors that are the most intense, like a bright primary color (imagine a bright red, blue, or yellow), have a high saturation value. You can get a better idea of these values by looking at the 3D color spaces pictured below.

Most of these different color spaces were either inspired by the human vision system, and/or developed for efficient use in television screen displays and computer graphics. You can read more about the history and the derivation of HLS and HSV color spaces here.

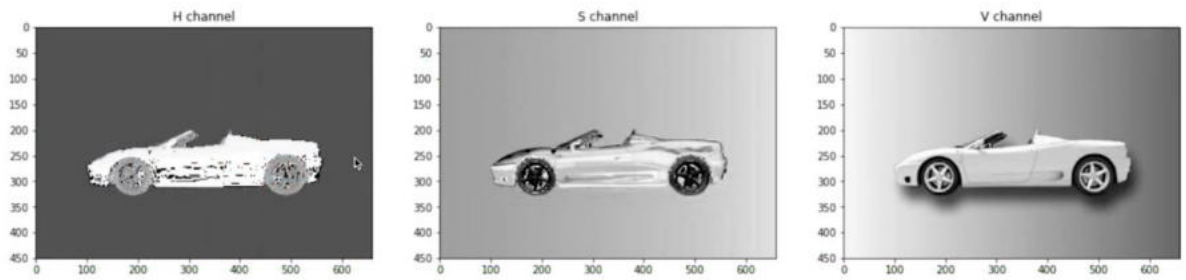


(Left) HSV color space, (Right) HLS color space

3.14 HSV Conversion

In the code example, I used HSV space to help detect a green screen background under different lighting conditions. OpenCV provides a function `hsv = cv2.cvtColor(im, cv2.COLOR_RGB2HSV)` that converts images from one color space to another.

After this conversion, I plotted the individual color channels; it was easy to see that the Hue channel remained fairly constant under different lighting conditions.



HSV colorspace

Import resources

In [15]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

import numpy as np
import cv2

%matplotlib inline
```

Read in RGB image

In [16]:
```
# Read in the image
image = mpimg.imread('images/car_green_screen2.jpg')

plt.imshow(image)
```

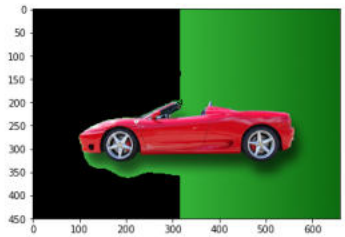
Out[16]: <matplotlib.image.AxesImage at 0x7f294776f9e8>

RGB threshold

Visualize the green threshold you defined in the previous, consistent green color case.

In [17]:
```
# Define our color selection boundaries in RGB values
lower_green = np.array([0,180,0])
upper_green = np.array([100,255,100])

# Define the masked area
mask = cv2.inRange(image, lower_green, upper_green)

# Mask the image to let the car show through
masked_image = np.copy(image)

masked_image[mask != 0] = [0, 0, 0]

# Display it!
plt.imshow(masked_image)
```

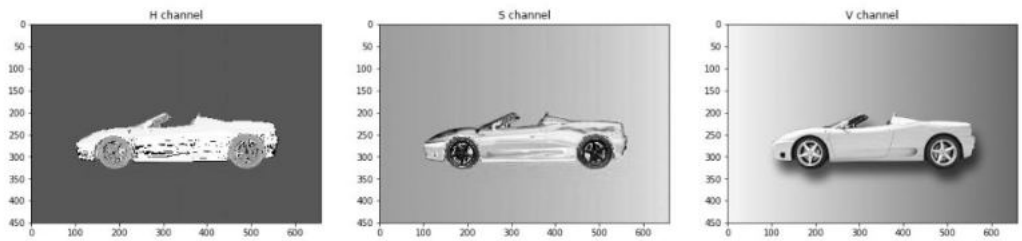
Out[17]: <matplotlib.image.AxesImage at 0x7f29477447f0>



Convert to HSV

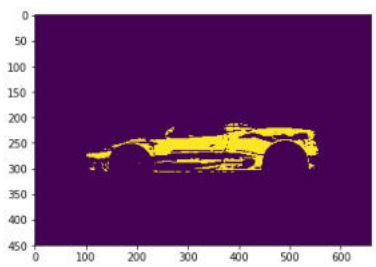
In [18]:
```
# Convert to HSV
hsv = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)

# HSV channels
h = hsv[:,:,0]
s = hsv[:,:,1]
v = hsv[:,:,2]

# Visualize the individual color channels
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,10))
ax1.set_title('H channel')
ax1.imshow(h, cmap='gray')
ax2.set_title('S channel')
ax2.imshow(s, cmap='gray')
ax3.set_title('V channel')
ax3.imshow(v, cmap='gray')
```

Out[18]: <matplotlib.image.AxesImage at 0x7f2945d6cba8>



TODO: Mask the green area using HSV color space

In [71]:
```
## TODO: Define the color selection boundaries in HSV values
#plt.imshow(h)
lower = np.array([0, 200, 0])
high = np.array([255, 255, 10])
#hi = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
##plt.imshow(hi)
m = cv2.inRange(hsv, lower, high)
plt.imshow(m)
masked_h = np.copy(hsv)
masked_h[m != 0] = 0
#v = cv2.cvtColor(masked_h, cv2.COLOR_HSV2RGB)
#plt.imshow(masked_h)
## TODO: Define the masked area and mask the image
# Don't forget to make a copy of the original image to manipulate
```



3.15 Day and Night Classification

You're on your way to being able to build classifier! You know how to analyze color and brightness in a given image, and that skill alone can help you distinguish between different traffic signs.

Now, I'm going to give you a classification challenge. What if I asked you to classify two types of images, taken during the day or at night when the sun has set, and I want you to separate these images into two classes: day or night.

day

night

This is actually an important classification challenge for self-driving cars. These cars need to know what kind of conditions they are driving in, so that they can safely navigate a road at any time of day, and so that they can still recognize other vehicles and surrounding objects whether it is dark or light outside!

We'll walk through each classification step together, but what do you think would be the first step in creating a classification model for day and night images?

Before you can classify any set of images, you have to look at them! **Visualizing the image data you're working with is the *first step* in identifying any patterns in image data and being able to make predictions about the data!**

So, we'll first load in this image data and learn a bit about the images we'll be working with.

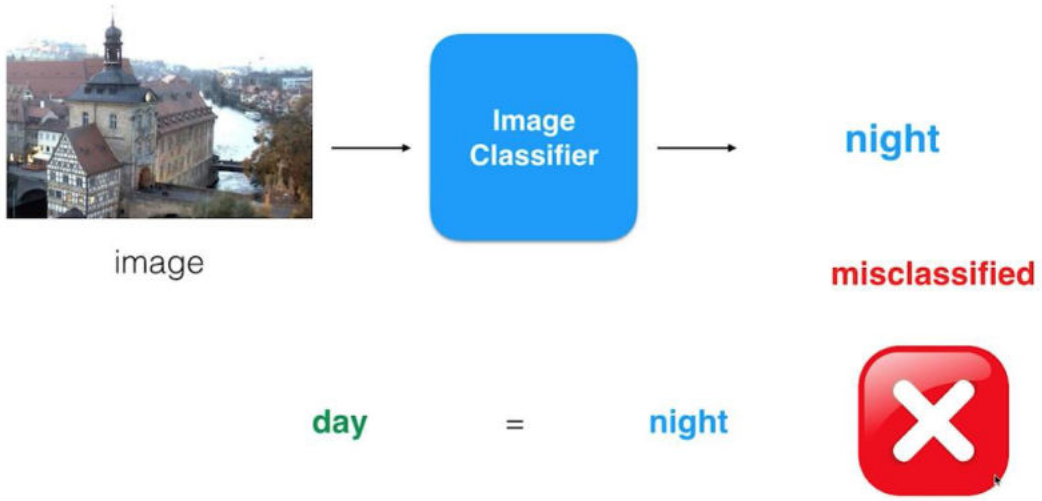
Why do we need labels?

You can tell if an image is night or day, but a computer cannot unless we tell it explicitly with a label!

This becomes especially important when we are testing the accuracy of a classification model.

A classifier takes in an image as input and should output a `predicted_label` that tells us the predicted class of that image. Now, when we load in data, like you've seen, we load in what are called the `true_labels` which are the *correct* labels for the image.

To check the accuracy of a classification model, we compare the predicted and true labels. If the true and predicted labels match, then we've classified the image correctly! Sometimes the labels do not match, which means we've misclassified an image.



3.16 Accuracy

After looking at many images, the accuracy of a classifier is defined as the number of correctly classified images (for which the predicted_label matches the true label) divided by the total number of images. So, say we tried to classify 100 images total, and we correctly classified 81 of them. We'd have 0.81 or 81% accuracy!

We can tell a computer to check the accuracy of a classifier only when we have these predicted and true labels to compare. We can also learn from any mistakes the classifier makes, as we'll see later in this lesson.

3.17 Numerical labels

It's good practice to use numerical labels instead of strings or categorical labels. They're easier to track and compare. So, for our day and night, binary class example, instead of "day" and "night" labels we'll use the numerical labels: 0 for night and 1 for day.

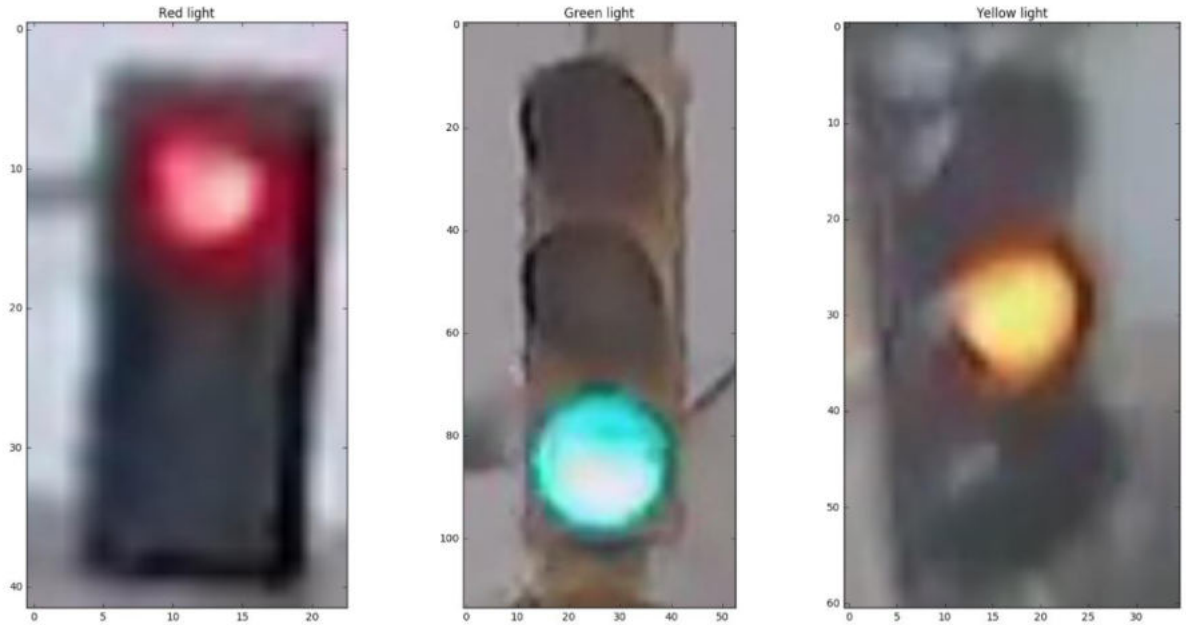
Okay, now you're familiar with the day and night image data AND you know what a label is and why we use them; you're ready for the next steps. We'll be building a classification pipeline from start to end!

Let's first brainstorm what steps we'll take to classify these images.

3.18 Project Overview

3.18.1 Traffic Light Classifier

In this project, you'll use your knowledge of computer vision techniques to build a classifier for images of traffic lights! You'll be given a dataset of traffic light images in which one of three lights is illuminated: red, yellow, or green.

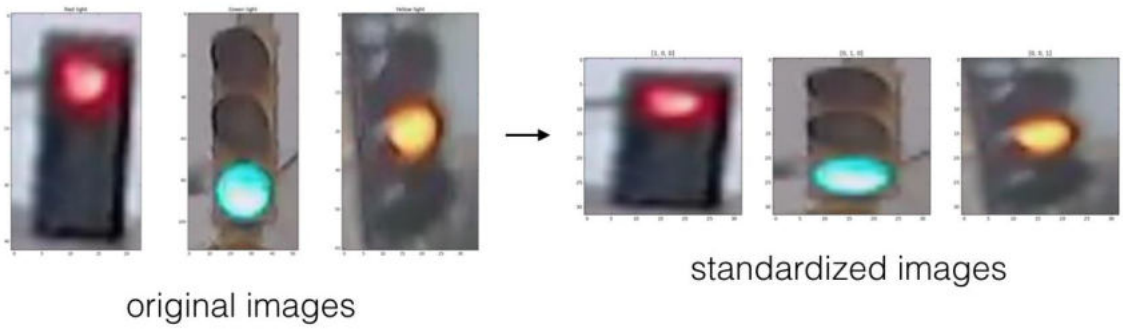


3.18.2 Classification Steps

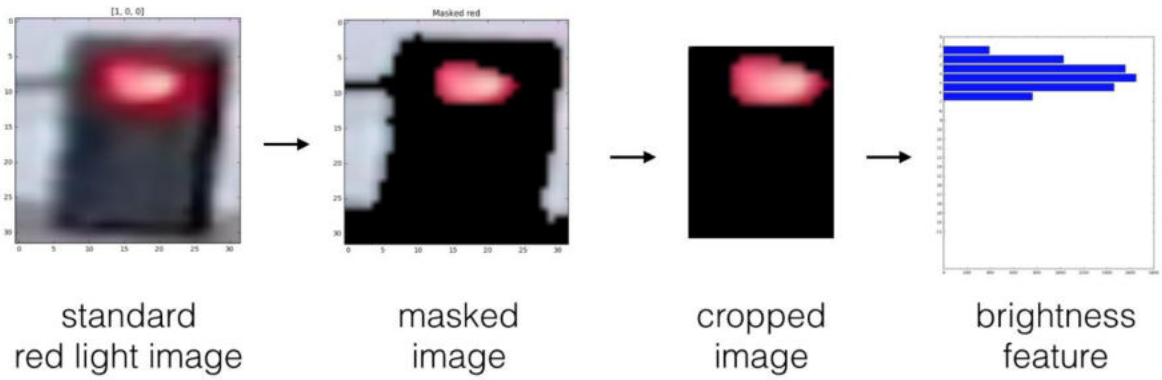
In the provided notebook, you'll pre-process these images, extract features that will help distinguish the different types of images, and use those features to classify the traffic light images into three categories: red, yellow, or green. The tasks will be broken down into a few sections:

1. Loading and visualizing the data. The first step in any classification task is to be familiar with your data; you'll need to load in the images of traffic lights and visualize them!

2. Pre-processing. The input images and output labels need to be standardized; that is, all the input should be of the same type of data and of the same size, and the output should be a numerical label. This way, you can analyze all the input images using the same procedures, and you know what output to expect when you eventually classify a new image.



3. Feature extraction. Next, you'll extract some features from each image that will be used to distinguish and classify these images. This is where you have a lot of creativity; features should be 1D vectors or even single values that provide some information about an image that can help classify it as a red, yellow, or green traffic light.



4. Classification and visualizing error. Finally, you'll write one function that uses your features to classify *any* traffic light image. This function will take in an image and output a label. You'll also be given code to classify a test set of data, compare your predicted label with the true label, and determine the accuracy of your classification model.

5. Evaluate your model. To pass this project, your classifier must be >90% accurate and never classify any red lights as green; it's likely that you'll need to improve the accuracy of your classifier by changing existing features or adding new features. I'd also encourage you to try to get as close to 100% accuracy as possible!

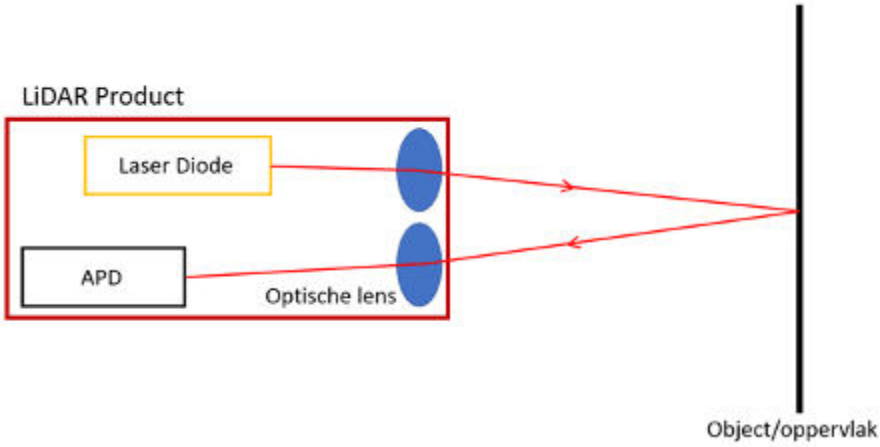
Chapter (4)

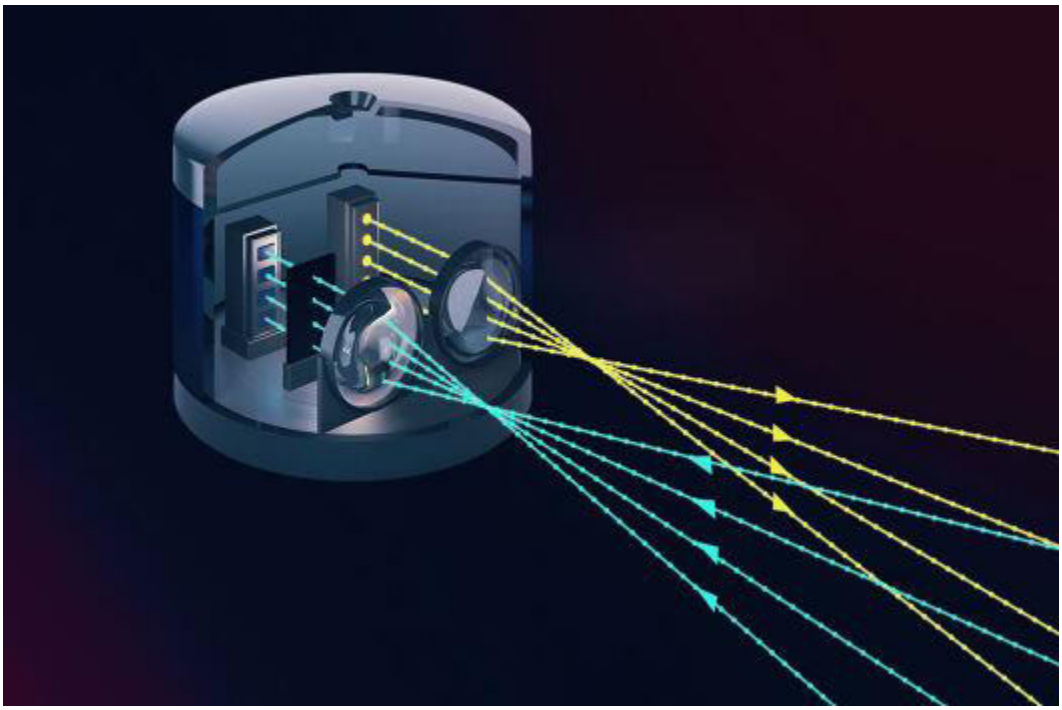
Distance Measurement

4.1 Introduction

To measure the distance of objects in the front, right, left, and back of the car, we can use many sensors, such ultrasonic sensor, LI-DAR, radar, and proximity, but each of them can detect a single beam on a single time (Figure 1), and if we need to take a frame of picture, and detect the depth-image of this picture, we have to rotate that beam in the (Φ, θ) system.

Another solution is to use multiple vertical beams (1 dimensional beam array) (Figure 2) and rotate that beam-array with 360 degrees around the z axis of the car, then detect the depth-image of all objects around the car, so these sensors, and it's installation on the car is costly, and need to be installed on the car roof which gives the car a bad view





Another solution is to use RGBD (Red, Green, Blue, and Depth) like the Kinect 1, but these cameras can't work in outdoors correctly, because it's work depends on transmitting an IR beam, then receive it and measure the distance between the camera, and the object by Eq. 1:
Speed (C) = distance / time.

Kinect2 working theory is transmitting an IR signal with a specific frequency, then receive it and measure the distance by the Eq. 1, so it can work in outdoors, all bodies have some absorption coefficient depends on the temperature of that body, and the color of it.

The RGBD Cameras is Expensive and it's price depends on the resolution of RGB Camera, the resolution of Depth Camera, it's accuracy, and it's size.

4.2 Human Vision System

If we tried to think enough about another ways to measure the distance between camera, and objects, the first thing we have to think about, is How we can expect the distance between us and some object, or more specifically, How can we see objects?

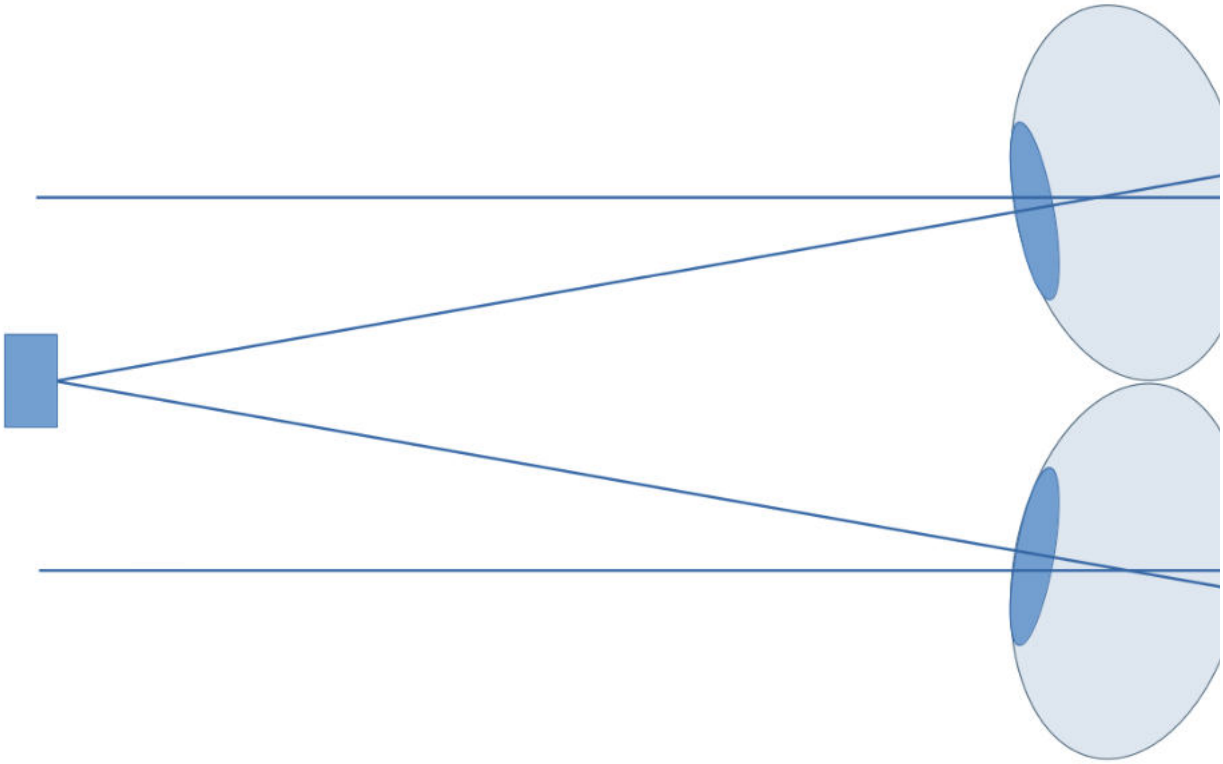
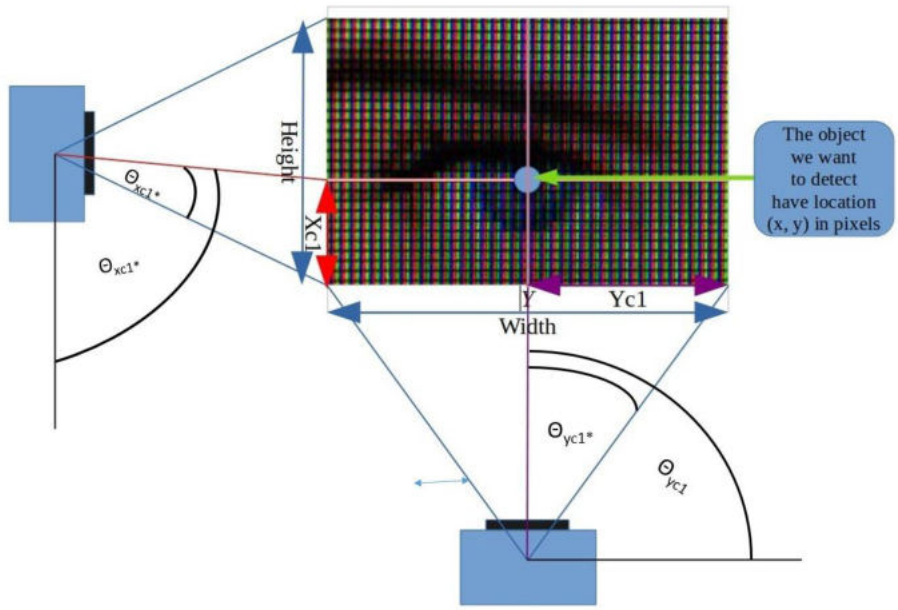


Figure 3

As we see, the optical axis of the right eye is intersects the optical axis of the left eye in a point which the object is located, then we can see the object.

And that's way that we can define some point, (two vector intersects themselves), and the intersection of them is a point.



And the optical axis of the eye is the Z axis, and the Z axis is a vector from the (retina) perpendicular on the (X, Y) plane.

That is the way that we can see and expect the distance from our eyes to some object, then let us try to see how camera works.

4.3 How Camera Works

The camera is a tool used to get a frame (picture) of a specific area, the picture consists of pixels, each pixel have some location (x, y) and values of (Red, Green, Blue) which make the color of the pixel (Figure 4).

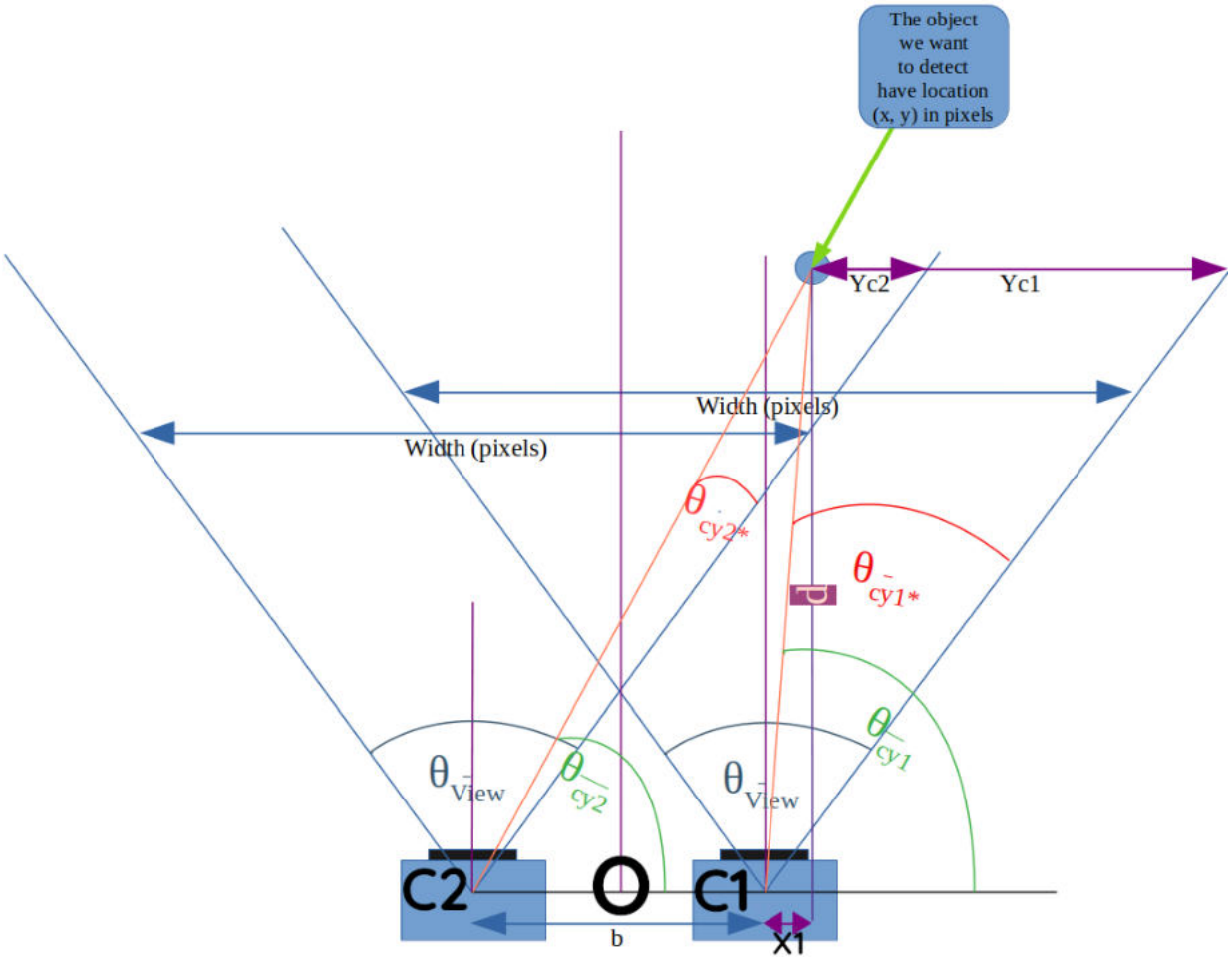
The location of the pixel could be considered as a vector from the sensor of camera to the Object.

So, If we used tow cameras (Figure 5) and we put them on same (Y) axis, and their optical axis is parallel, with distance (b) between them, and they have same **known** (θ_{View_x}, θ_{View_y}) the same **known** resolution, and the same focal length, then we can measure distance between the middle point between cameras (O), and the object.

4.4 Calculate Distance from RGB Camera

To simplify this operation, we will use the projections on ZY plane and ZX Plane, then combine them together to get the final values.

4.4.1 The projection on ZY Plane (Figure 5):



We need to measure θ_{cy1*} and θ_{cy2*} first to measure θ_{cy1} and θ_{cy2} .

To measure θ_{cy1*} , we need to see the width of pictures from a different way, the width of camera can represent the θ_{view_y} not in degrees, but in pixels, then the width of our picture which is the width of resolution of our camera in pixels is represents the θ_{view_y} of our camera.

Then,

θ_{cy1*} = (Yc1/width) * θ_{view_y}

$\theta_{cy2*} = (Yc2/width) * \theta_{view_y}$

Then, we can measure θ_{cy1} and θ_{cy2} by:

$\theta_{cy1} = ((180 - \theta_{view_y}) / 2) + \theta_{cy1*}$

$\theta_{cy2} = ((180 - \theta_{view_y}) / 2) + \theta_{cy2*}$

Then:

$\tan(\theta_{cy1}) = d/x1 \Rightarrow x1 = d/\tan(\theta_{cy1})$

$\tan(\theta_{cy2}) = d/(b + x1) \Rightarrow d = b*\tan(\theta_{cy2}) + x1*\tan(\theta_{cy2}) = b*\tan(\theta_{cy2}) + (d*\tan(\theta_{cy2})/\tan(\theta_{cy1}))$

and finally: $[d = b * \tan(\theta_{cy2}) / (1- (\tan(\theta_{cy2})/\tan(\theta_{cy1})))]$

which is the Z coordinate of our object.

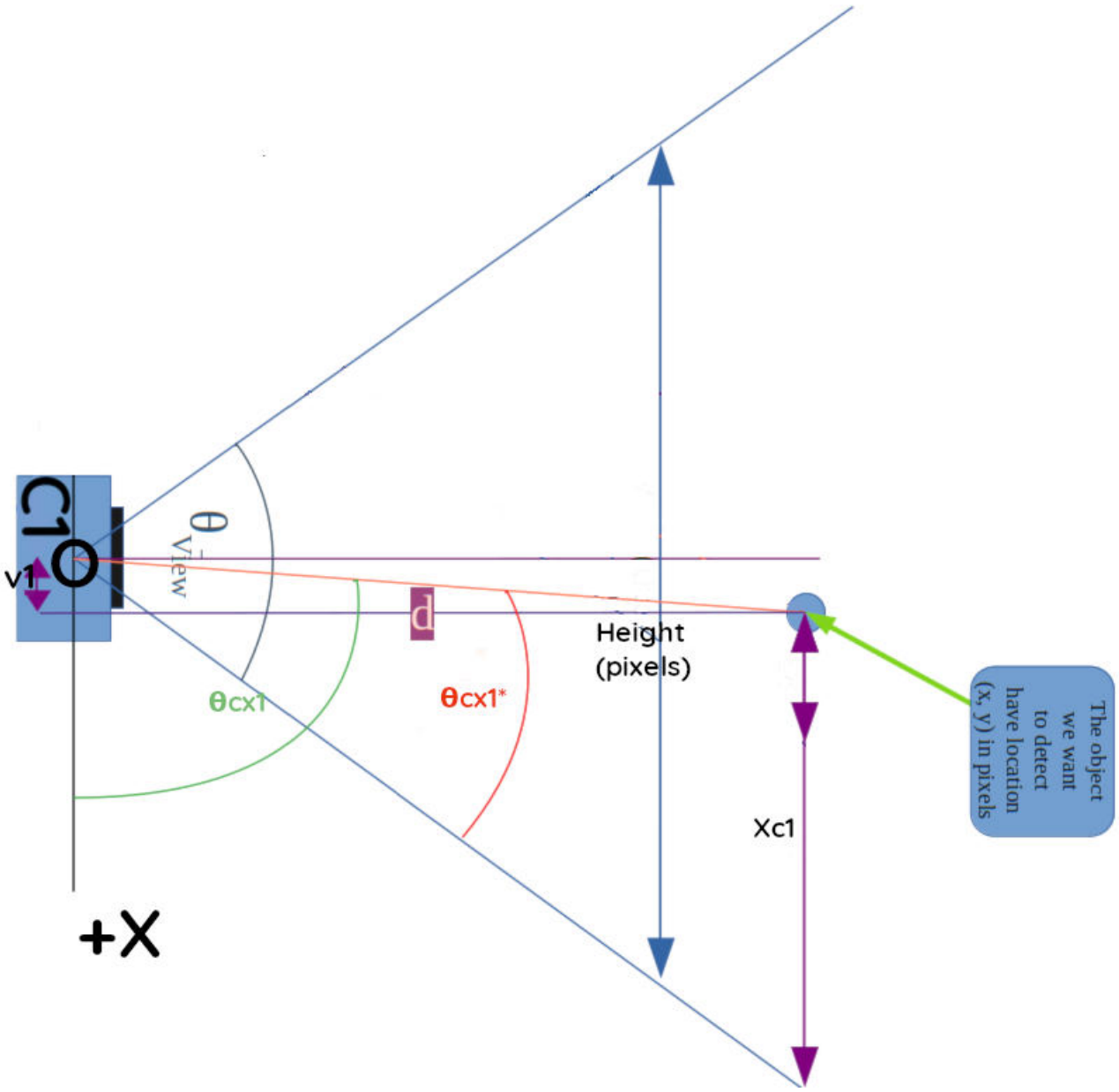
Then if we need to measure the (Y) coordinate of the object, we need first to get x1:

$x1 = d/\tan(\theta_{cy1})$, and the Y coordinate is equal to (b/2 + x1).

Finally, we got the Z, and Y coordinates of our object.

4.4.2 The projection on ZX Plane (Figure 6):

From the projection on ZY plane, we have measured the Z coordinate which is (d) notation on Figure 6, and we need to measure the v1 distance to get the X coordinate.



$\tan(\theta_{cx1}) = d/v1 \Rightarrow v1 = d/\tan(\theta_{cx1})$

And:

$\theta_{cx1} = ((180 - \theta_{view_x}) / 2) + \theta_{cx1*}$,

$\theta_{cx1*} = (Xc1/height) * \theta_{view_x}$

Then finally, the X coordinate of our object is measured from the formula:

$X = v1$

Then, We now have the coordinates of our object, (X, Y, Z), and if we need to measure the absolute distance between the point O, we can measure it by:

$D = sqrt(X^2 + Y^2 + Z^2)$: D is the absolute distance between O and the object

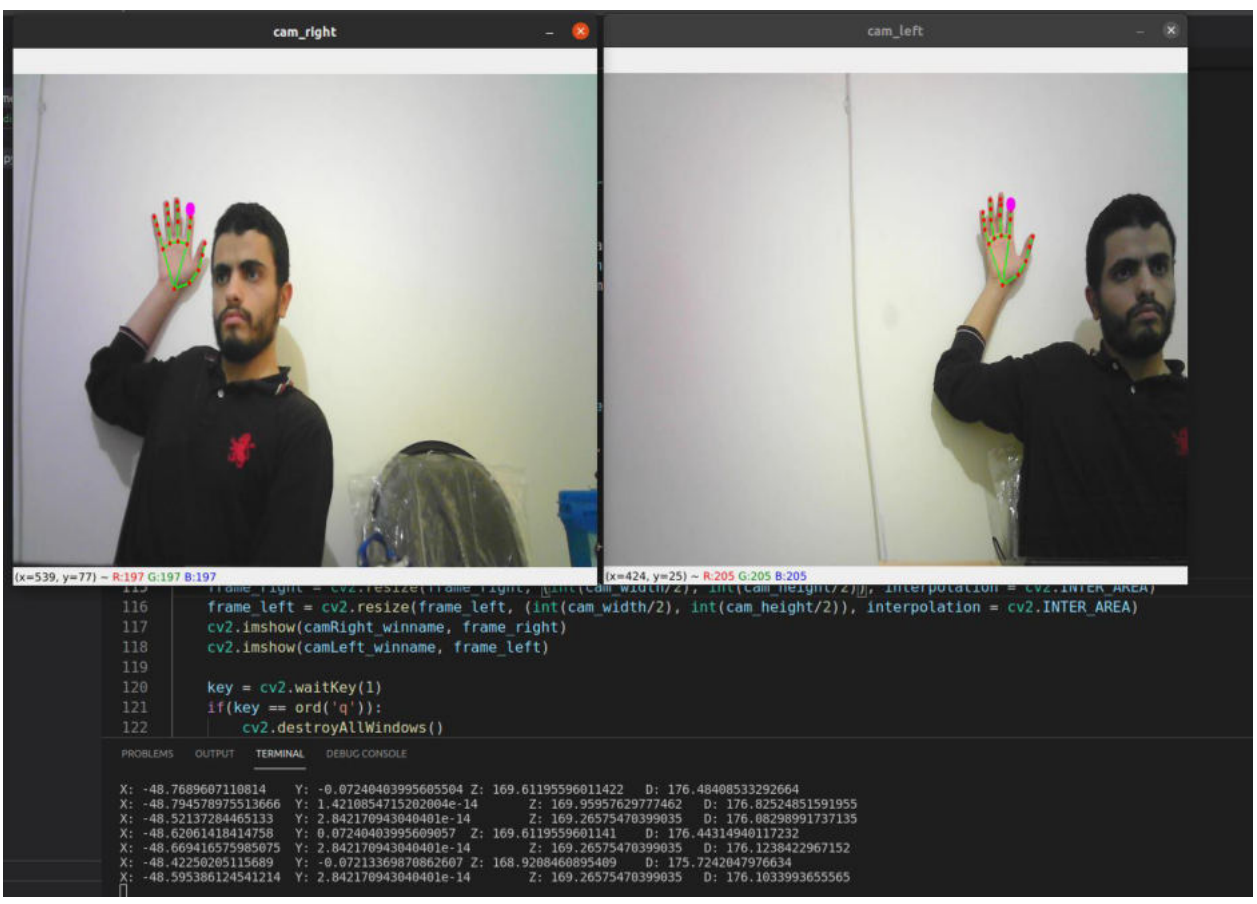
4.5 Measurement Code

```
import mediapipe as mp
import cv2
import math

b = 70.8
view_th_y = 60
view_th_x = 50

cam_width = 1280
cam_height = 1080

cam_right = '/dev/video2'
cam_left = '/dev/video4'

cap_right = cv2.VideoCapture(cam_right)
cap_left = cv2.VideoCapture(cam_left)

cap_right.set(cv2.CAP_PROP_BUFFERSIZE, 1)
cap_left.set(cv2.CAP_PROP_BUFFERSIZE, 1)

cap_right.set(3, cam_width)
cap_right.set(4, cam_height)
cap_left.set(3, cam_width)
cap_left.set(4, cam_height)

mpHands = mp.solutions.hands
hands   = mpHands.Hands(False, 2, 0.5, 0.5)
mpDraw  = mp.solutions.drawing_utils


camRight_winname = "cam_right"
camLeft_winname = "cam_left"
```

Figure 7: initializing important variables

```python
def get_distance(cx_right, cy_right, cx_left, cy_left):
    if(cx_right == 0 or cx_left == 0):
        return 0, 0, 0, 0
    else:
        th_x_right = ((180 - view_th_x)/2) + ((cam_height - cy_right)/cam_height) * view_th_x
        th_y_right = ((180 - view_th_y)/2) + ((cam_width - cx_right)/cam_width) * view_th_y
        th_x_left = ((180 - view_th_x)/2) + ((cam_height - cy_left)/cam_height) * view_th_x
        th_y_left = ((180 - view_th_y)/2) + ((cam_width - cx_left)/cam_width) * view_th_y

        th_x = (th_x_right + th_x_left ) / 2

        th_x = th_x * math.pi / 180
        th_y_right = th_y_right * math.pi / 180
        th_y_left  = th_y_left * math.pi / 180

        th_x_deg        = th_x * 180/math.pi
        th_y_right_deg  = th_y_right * 180/math.pi
        th_y_left_deg   = th_y_left * 180/math.pi

        tan_th_x        = math.tan(th_x)
        tan_th_y_right  = math.tan(th_y_right)
        tan_th_y_left   = math.tan(th_y_left)

        Z = (b * tan_th_y_left) / (1 - (tan_th_y_left/tan_th_y_right))
        Y = (b/2) + Z / tan_th_y_right
        X = Z / tan_th_x

        D = math.sqrt((X**2) + (Y**2) + (Z**2))

        return X, Y, Z, D
```

Figure 8: distance measurement function



Chapter (5)

Microcontroller

5.1 Introduction:

World of Microcontroller is large and forked depend on Memory Architecture, Instructions Set, Bit-Bus…etc.

ARM, AVR, Motorola, PIC, Texas…etc. is Types of Microcontrollers, each of them has

Specifications were different from the others.

To choose the best Microcontroller, you need to specify Data-Bus, Memory, Communications Protocols, Type and Numbers of Interrupts and if you need PWM, Frequencies of Cristal, No. of I/O RISC.

The AT90S8515 was the first microcontroller that was based on AVR architecture however the first microcontroller to hit the commercial market was AT90S1200 in the year 1997.

5.1.1 AVR families:

Tiny AVR (ATtinyXXXX): Less memory, small size, suitable only for simpler applications

Mega AVR (ATmegaXXXX): These are the most popular ones having a good amount of memory (up to 256 KB), a higher number of inbuilt peripherals and suitable for moderate to complex applications.

Xmega AVR (ATXmegaXXXX): Used commercially for complex applications, which require large program memory and high speed. And other Specifications give you the best performance for your project.One of the Big families in this world is the AVR family, they have a lot of Microcontrollers with different Specifications for letting your choice is the Best option you Need

5.1.2 History of AVR:

AVR was developed in the year 1996 by Atmel Corporation. The architecture of AVR was developed by Alf-Egil Bogen and Vegard Wollan. AVR derives its name from its developers and stands for Alf-Egil Bogen Vegard Wollan RISC microcontroller, also known as Advanced Virtual RISC. The AT90S8515 was the first microcontroller that was based on AVR architecture however the first microcontroller to hit the commercial market was AT90S1200 in the year 1997.

Series Name	Pins	Flash Memory	Special Feature
TinyAVR	6-32	0.5-8 KB	Small in size
MegaAVR	28-100	4-256KB	Extended peripherals
XmegaAVR	44-100	16-384KB	DMA , Event System included

The family we will use is Mega AVR, specifically: Atmega32 and Atmega328.

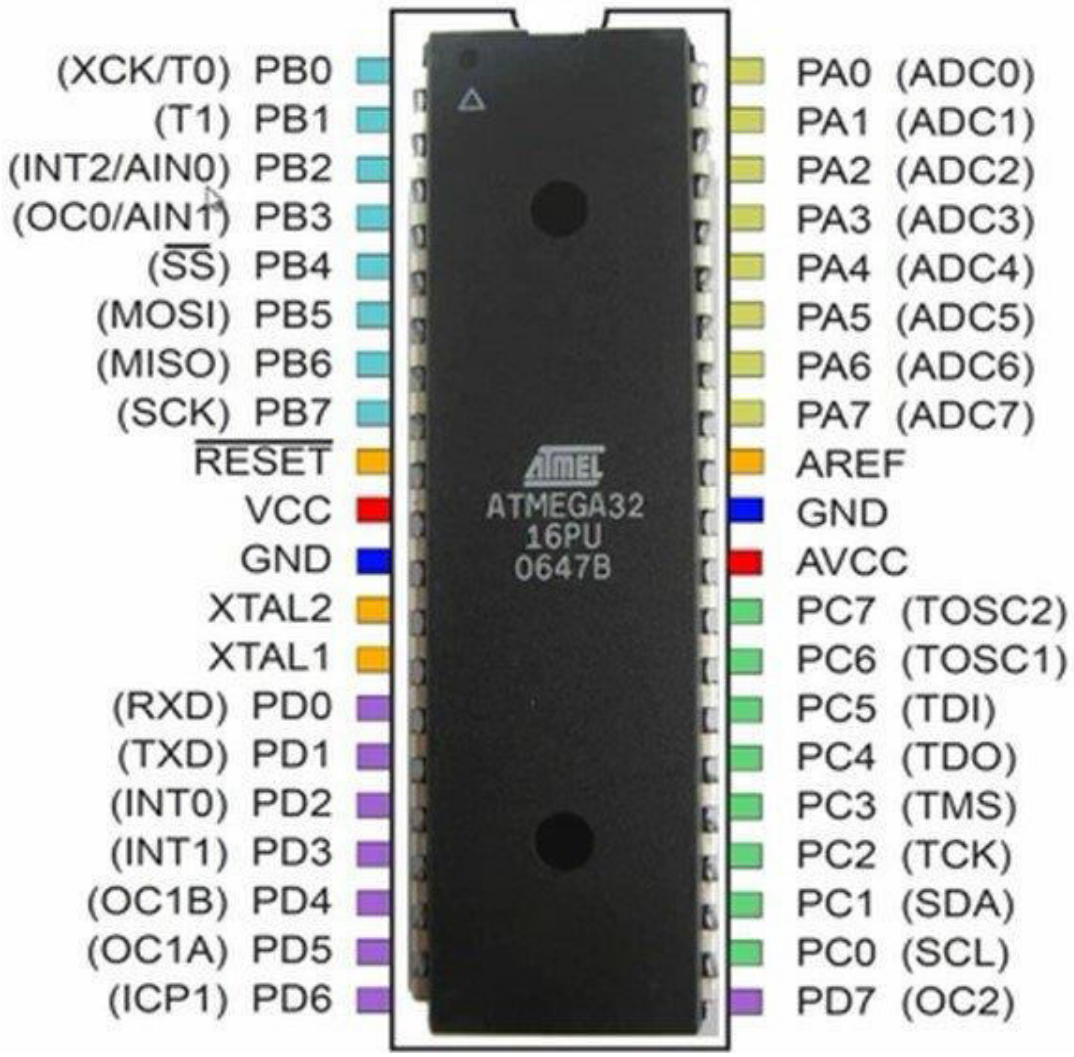
5.2 The Atmega32

5.2.1 Introduction:

ATmega32 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. AVR can execute 1 million instructions per second if the cycle frequency is 1MHz.

Atmega32 is equipped with 32 general-purpose registers which are coupled directly with the Arithmetic Logical Unit (ALU) of CPU.

Atmega32 PDIP has: 40 Pins, 32 for general-purpose registers, 6 for Power, 2 for External crystal oscillator.



5.2.2 Specification:

32 x 8 general working purpose registers.

32K bytes of in-system self-programmable Flash program memory

2K bytes of internal SRAM

1024 bytes EEPROM

Available in 40 pin DIP.

32 programmable I/O lines

8 Channel, 10 bit ADC

Two 8-bit timers/counters with separate prescale and compare modes

One 16-bit timer/counter with separate Prescaler, compare mode, and capture mode.

4 PWM channels

In system programming by an on-chip boot program

Programmable watchdog timer with a separate on-chip oscillator.

Programmable Serial USART

Master/slave SPI serial interface

Six sleep modes: Idle, ADC noise reduction, power-save, power-down, standby, and extended standby.

Internal calibrated RC oscillator

External and internal interrupt sources

Power-on reset and programmable brown-out detection.

ATmega32 is a powerful microcontroller because of its in-system self-programmable Flash on a monolithic chip, which provides a highly flexible and cost-effective solution to many embedded control applications.

5.2.3 Pin Descriptions:

VCC: Digital voltage supply

GND: Ground

Port A (PA7-PA0): This port serves as analog inputs to the A/D converter. It also serves as an 8-bit bidirectional I/O port if the A/D converter is not used.

Port B (PB7-PB0) & Port D (PD7-PD0): It is an 8-bit bidirectional I/O port. Its output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, these are extremely pulled low if the pull-up resistors are activated. It also serves various special function features of the ATmega32.

Port C (PC7-PC0): It is an 8-bit bidirectional I/O port. If the JTAG interface is enabled, the pull-up resistors on pins PC5 (TDI), PC3 (TMS), and PC2 (TCK) will be activated.

Reset: It is input.

XTAL1: It is an input to the inverting oscillator amplifier and input to the internal clock operating circuit.

XTAL2: It is an output from the inverting oscillator amplifier.

AVCC: It is the supply voltage pin for Port A and A/D converter. It should be connected to VCC.

AREF: AREF is the analog reference pin for the A/D converter.

5.2.4 Flash Program Memory:

ATmega32 contains 32 Kbytes on-chip in-system reprogrammable flash memory for program storage. Flash is organized as 16k X 16 and its memory is divided into two sections Boot program section and the application program section.

5.2.5 SRAM Data Memory:

The Register file, I/O memory, and the internal data SRAM are addressed by the lower 2144 data memory locations. The first 96 locations address the Register file and I/O memory, and the internal data SRAM is addressed by the next 2048 locations. Direct, indirect with displacement, indirect, indirect with pre-decrement, and in-direct with post decrement are the 5 different addressing modes for the data memory covering. The 32 general-purpose registers, 64 I/O registers, and 2048 bytes of internal data SRAM are accessible by using these addressing modes.

5.2.6 EEPROM Data Memory:

It contains 1024 bytes of data EEPROM memory. It can be accessed as a separate data space in which single bytes can be read and written.

5.2.7 I/O Memory:

All I/Os and peripherals are placed in the I/O space. The I/O locations are accessed by the IN and OUT instructions, transferring the data between the 32 general-purpose registers and the I/O space.

Chapter (6)

The Software

6.1 Software Architecture

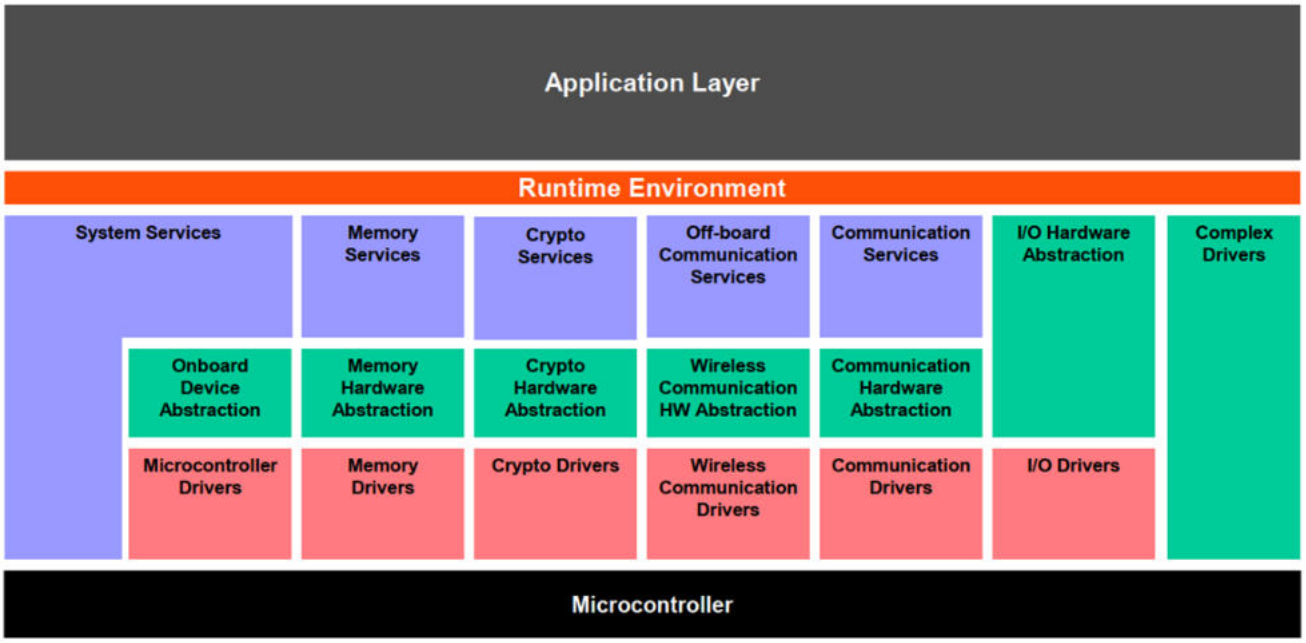
6.1.1 Introduction:

For a standard of AUTOSAR "Automotive Open System Architecture", there are many of layers Architecture for strong and stable systems of automotive electronic control units (ECUs).

The primary goal of the AUTOSAR partnership is the standardization of a common methodology, basic system functions, and functional interfaces. This enables development partners to integrate, exchange, re-use, and transfer functions within a vehicle network and substantially improves their efficiency of development. Having this goal in mind, AUTOSAR pushes the paradigm shift from an ECU-based to a function-based system design attempt in automotive software development and enables the management of the ever-growing software and E/E complexity concerning technology and economics.

Layers:

1. Microcontroller abstraction layer "MCAL"
2. Hardware abstraction layer "HAL"
3. System service layer "SSL"
4. Application layer "APL"



- AUTOSAR Confidential -

In our project we build Drivers at a specific layer:

7.1.2 Microcontroller Abstraction Layer (MCAL):

ATMEGA32A	ATMEGA328P-AU
DIO "Digital Input Output"	DIO
PWM0 "Pulse Width Modulation"	PWM0,PWM1,PWM2
PWM2	USART , I2C
UART "Universal asynchronous receiver-transmitter"	Timer0,1,2

7.1.3 Hardware Abstraction Layer (HAL):

ATMEGA32A	ATMEGA328P-AU
Motor Driver	MPU6050
LEDs Driver	GPS
RGB Driver	Ultrasonic
	Bluetooth

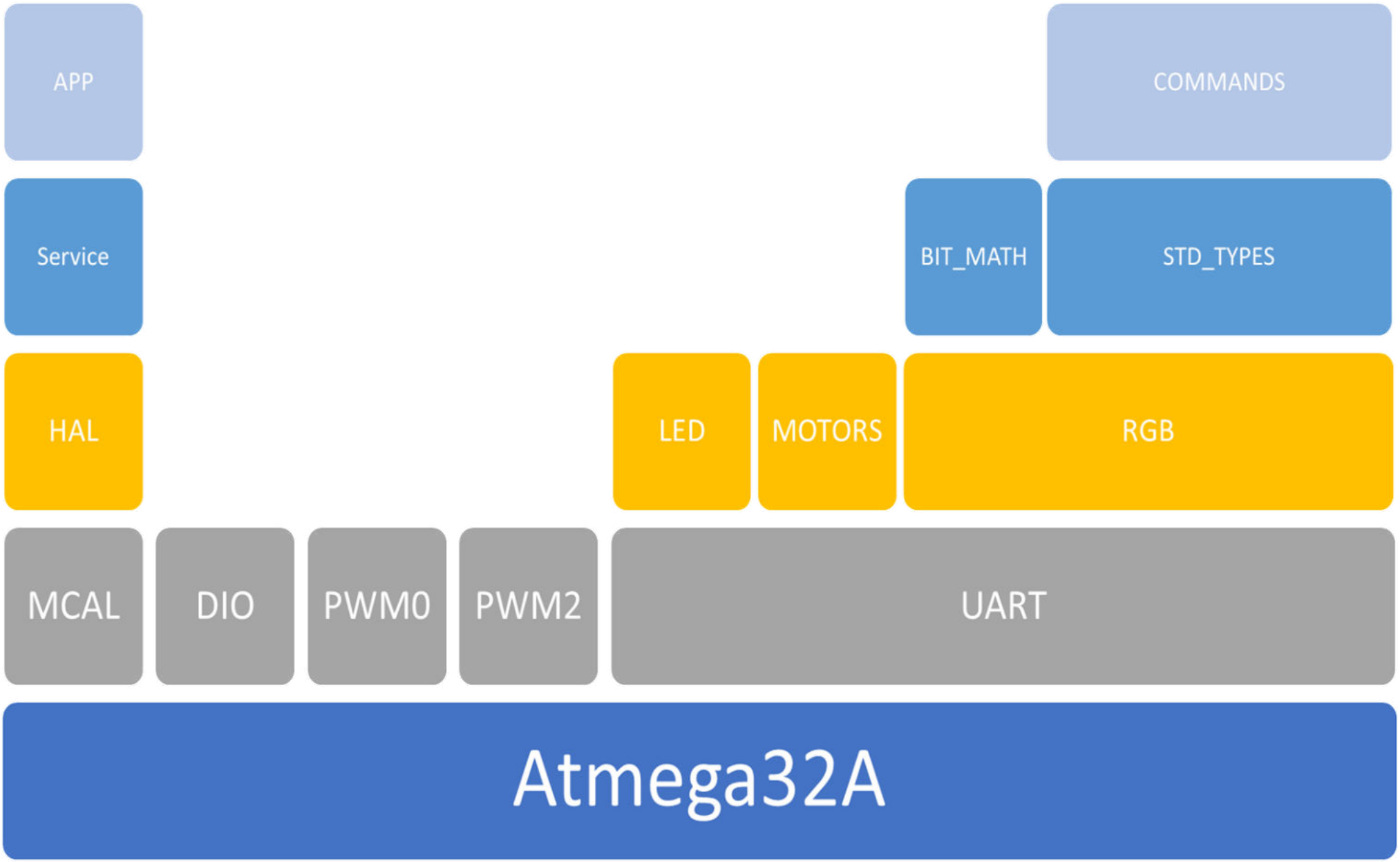
7.1.4 Services:

ATMEGA32A	ATMEGA328P-AU
Standard types (STD_TYPES)	
Bit Math (BIT_MATH)	

7.1.5 Application

ATMEGA32A	ATMEGA328P-AU
Command Check	Battery Management Module (calculate battery charge state)
Main Super loop	Serial Communication (send – receive)
Peripheral Initialization	Accelerometer and Gyroscope readings collection
RGB Color Select	Bluetooth module commands identification
LEDs Control	GPS reading processing
Serial Communication (send – receive)	Distance Measurement
Speed control	Battery Management Module (calculate battery charge state)
Direction control	10 Hz Scheduler (data transmission).

7.1.6 Microcontroller Layered Architecture


APP
COMMANDS
Service
BIT_MATH
STD_TYPES
HAL
LED
MOTORS
RGB
MCAL
DIO
PWM0
PWM2
UART
Atmega32A

7.2 Naming Conventions

For Readable and Repairable Code, we following the standard for Naming variables and functions and any things in our code.

7.2.1 Data Types:

U: Unsigned

S: Signed

8, 16, 32, 64: No. of Bits

enum: an enumerated type.

7.2.2 Local Variables:

LOC_

7.2.3 Global Variables:

GLO_

7.2.4 Passing Type:

Copy: is passing by value

Ptr: passing by Reference with Pointer.

7.2.5 Name of Variable:

Start with LOC/GLO, then (_), the size of the variable, Name of the variable.

The Name should be Clear, meaningful, Easy to Remembering and using.

EX: Copy_u8Direction

It's passing a variable by value, have 8 Bit size, to Set Direction anyhow.

7.2.6 Name of Function:

NAME OF DRIVER_Returned Data Type Name of Function "Clear, meaningful, Easy to Remembering and using"

EX: DIO_enumSetPortValue

It's DIO Driver, expected enum as Returned Data Type, and from naming we can guess it's Set Value for one of the Ports of DIO "**HIGH** or **LOW**".

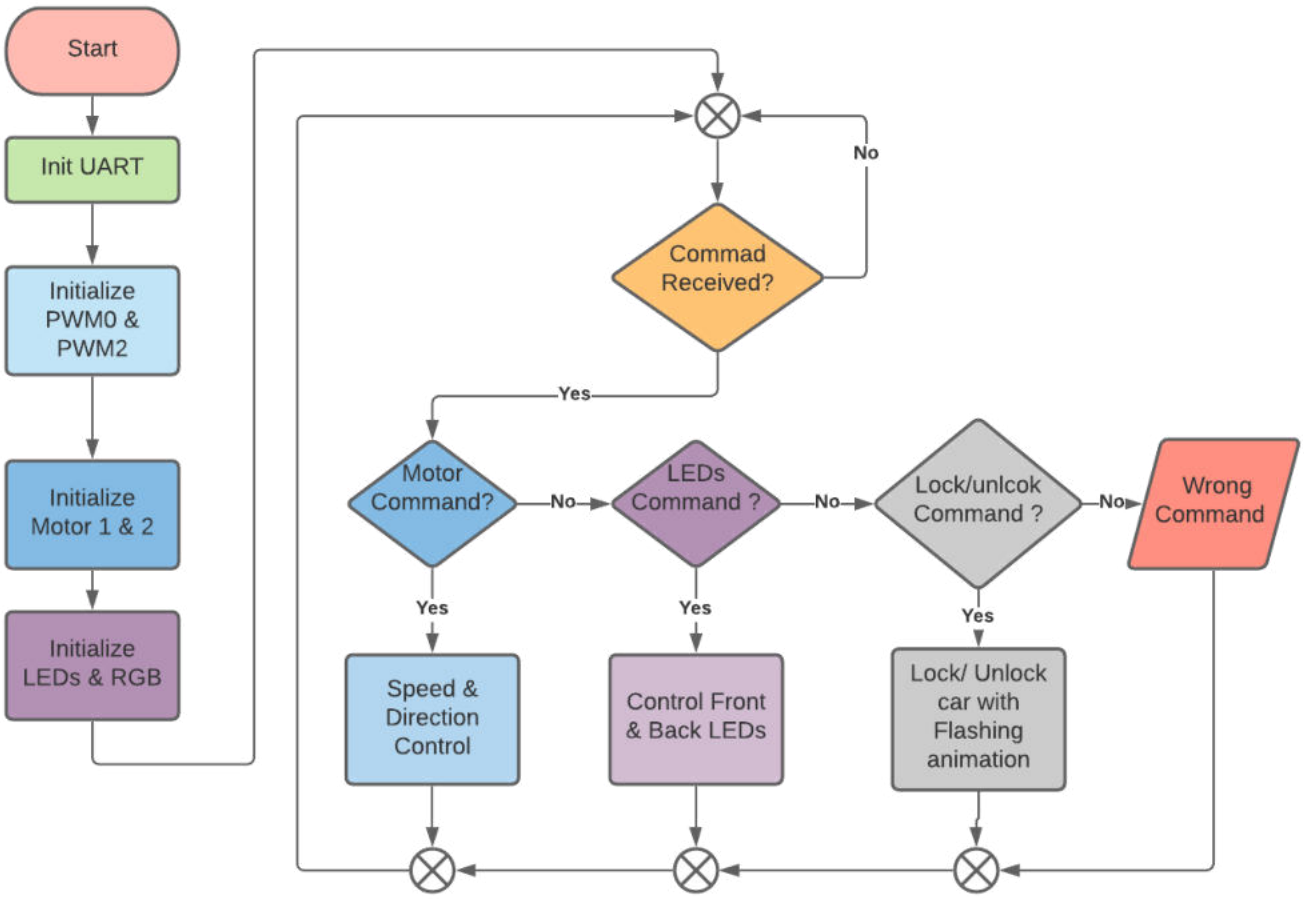
7.3 Commands

M2F	Set Back Motor Forward
M2B	Set Back Motor Back
M2S	Set Back Motor Stop
M2X	Set Back Motor Break
M2V	Set New Speed for Back Motor
M1R	Set Front Motor Right
M1L	Set Front Motor Left
M1S	Set Front Motor Stop
M1X	Set Front Motor Break
M1D?	Asking for Direction (R or L)
COF	Lock Car Off (Car OFF)
CON	Unlock Car (Car ON)
LFOR	Turn ON Light of Front Right

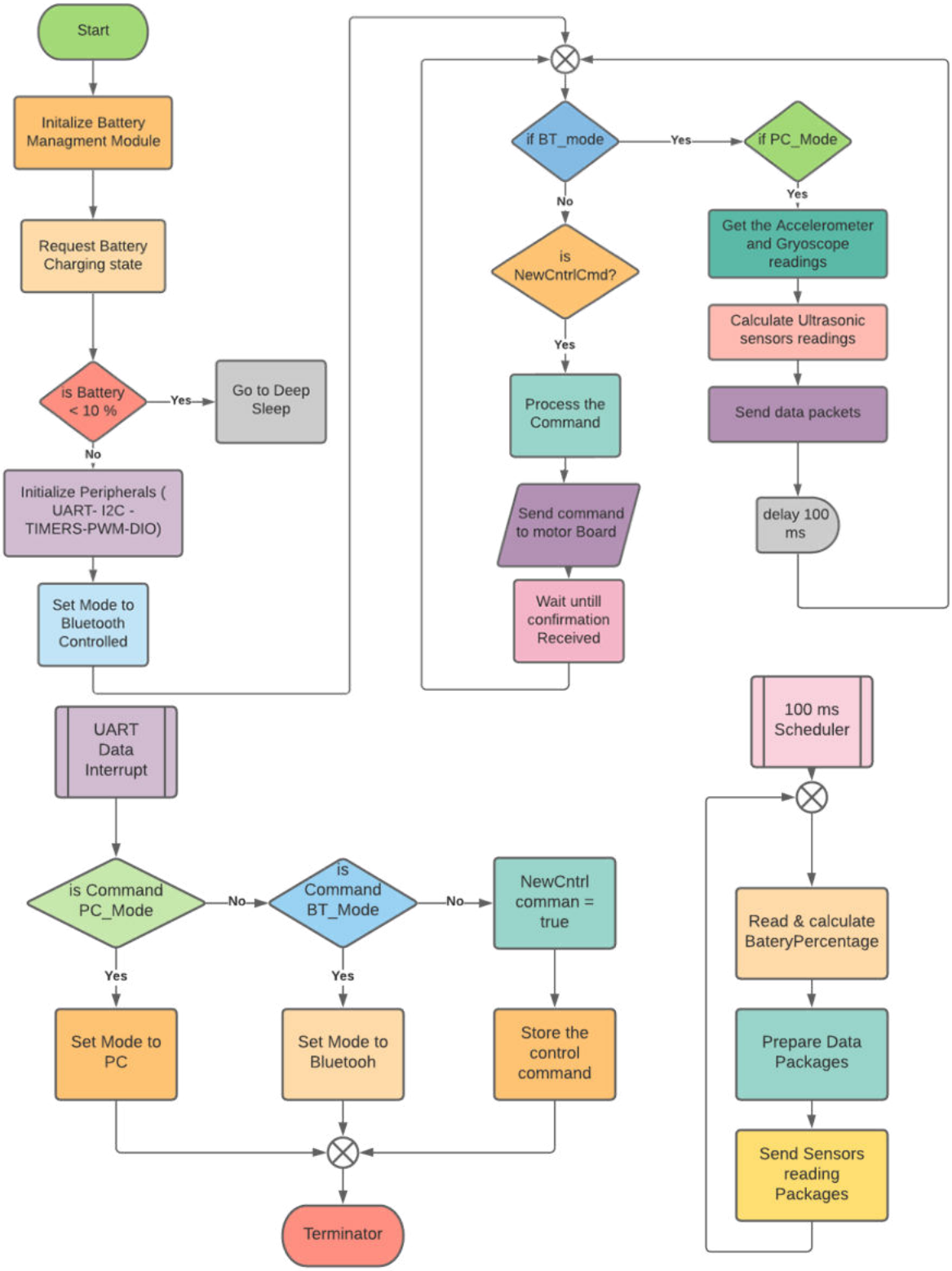
LFOL	Turn ON Light of Front Left
LFFR	Turn OFF Light of Front Right
LFFL	Turn OFF Light of Front Left
LBOR	Turn ON Light Of Back Right
LBOL	Turn ON Light Of Back Left
LBFR	Turn OFF Light Of Back Right
LBFL	Turn OFF Light Of Back Left

7.4 Flow Chart

7.4.1 Motor MCU Flow Chart



7.4.2 Main MCU Flow Chart



Chapter (7)

Hardware

In this chapter, we are going to discuss the hardware components used to apply the software and algorithms mentioned in the previous chapters.

8.1 Car body

We Recycled a Toy Car Chasse of Scale 1:8 of Hummer H2 Model Car and we use it as our frame. The Car frame offered us plenty of space to work on and mount the components on it.

8.2 Control Boards

The control boards is divided into two functional blocks

A- Main Controller

Collect the sensors data and communicate with Raspberry Pi to send collected data and get the control requests

B- Motor Controller

Control the motors and the lights of the car

8.3.1 Main Board Schematic

Main board collect most of the sensors in the car which include (MPU6050, GPS and Ultrasonic HC-SR04)

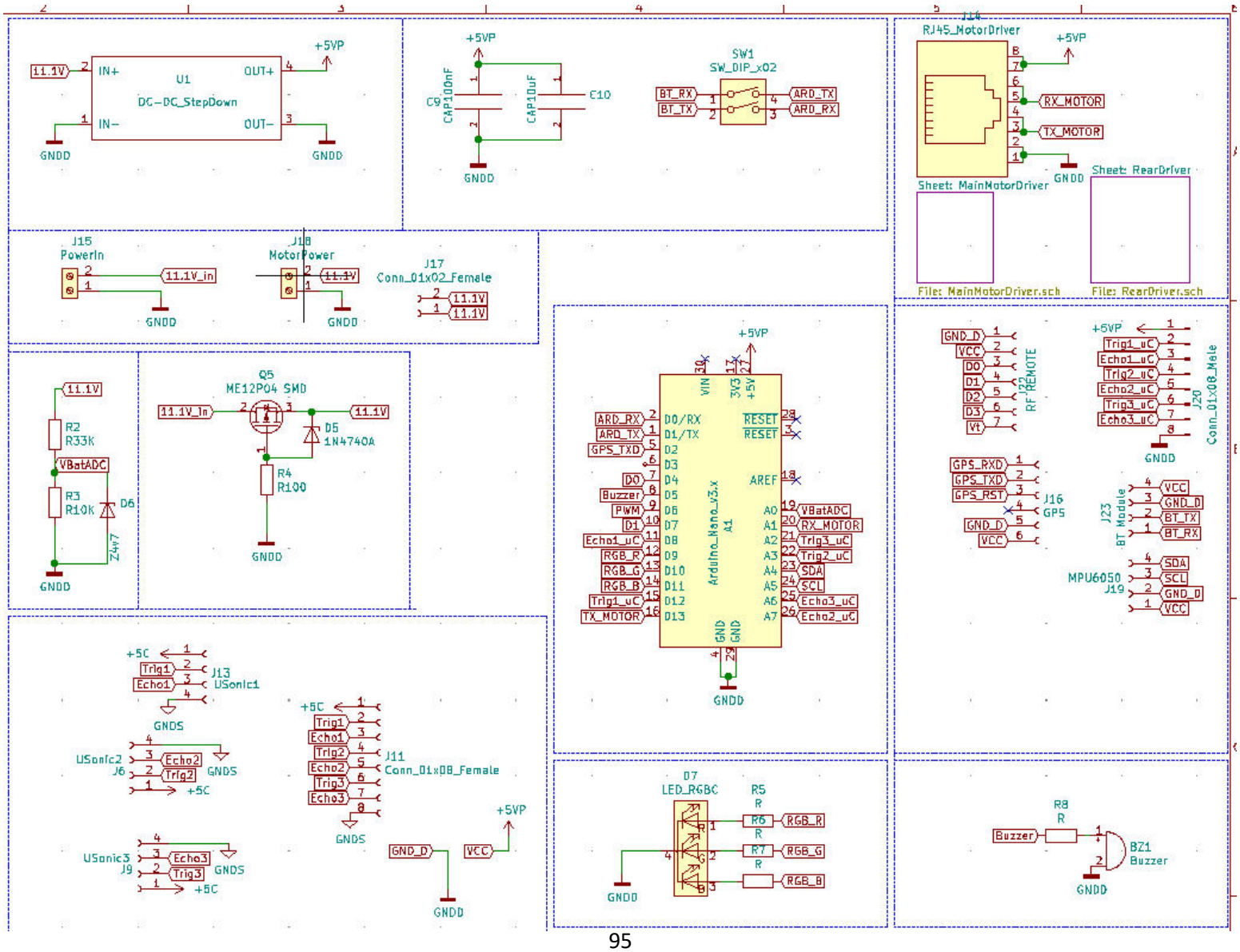
Features:

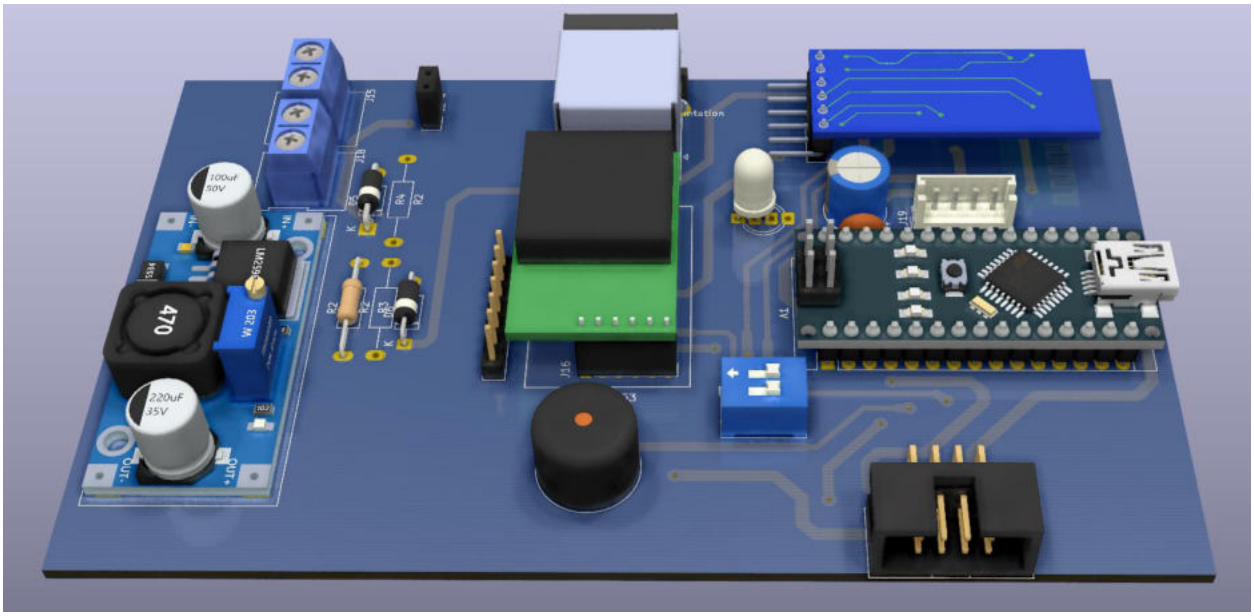
1- Reverse Polarity Protection, as it distribute the Power to the whole car

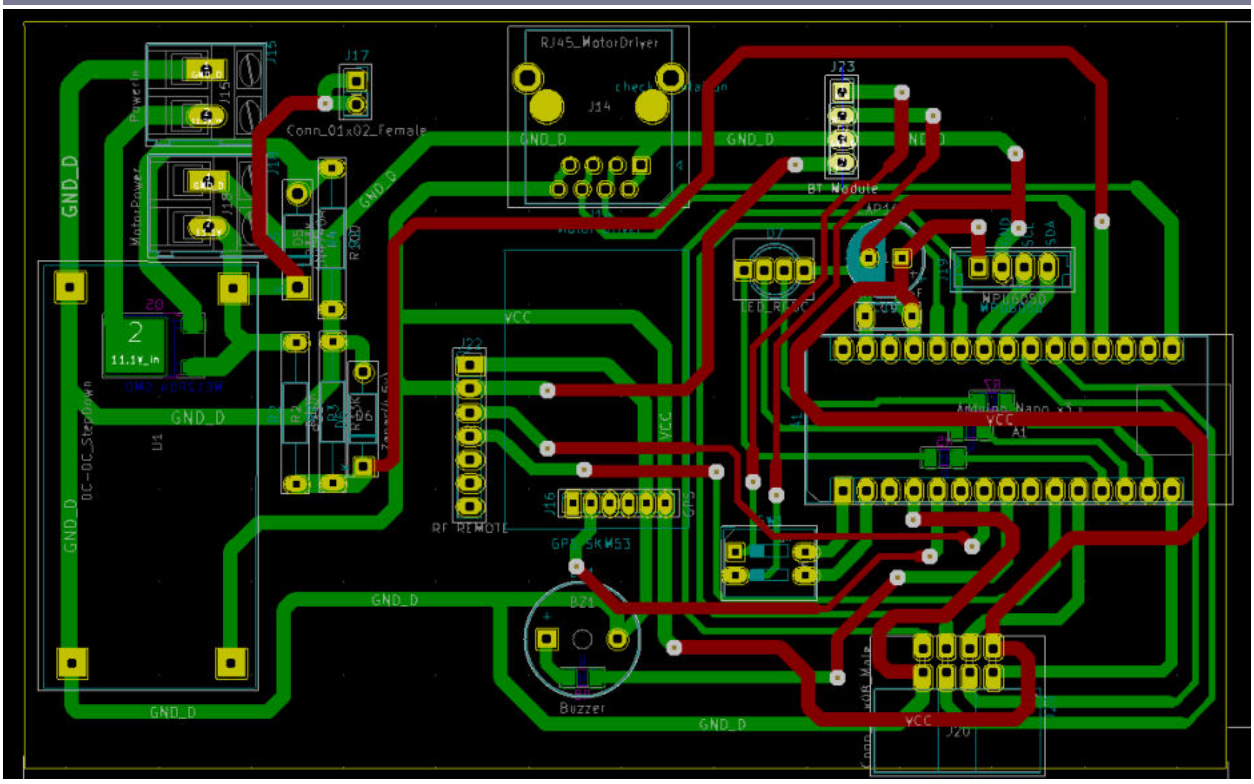
2- Step down converter 5v/3A to power all boards and small steering motor

3- Continuous Battery Level Measurement

4- RGB as Status LED & RF Remote to Lock and Unlock the Car




100uF
50V
470
220uF
35V
OUT-
OUT+


RJ45_MotorDriver
J14
J17
Conn_01x02_Female
J23
BT Module
GND_D
VCC
J22
RF REMOTE
J16
BZ1
Buzzer
J20
11.1V_in
DC-DC_Stepdown
U1

8.3.2 Motor Control Board

Features:

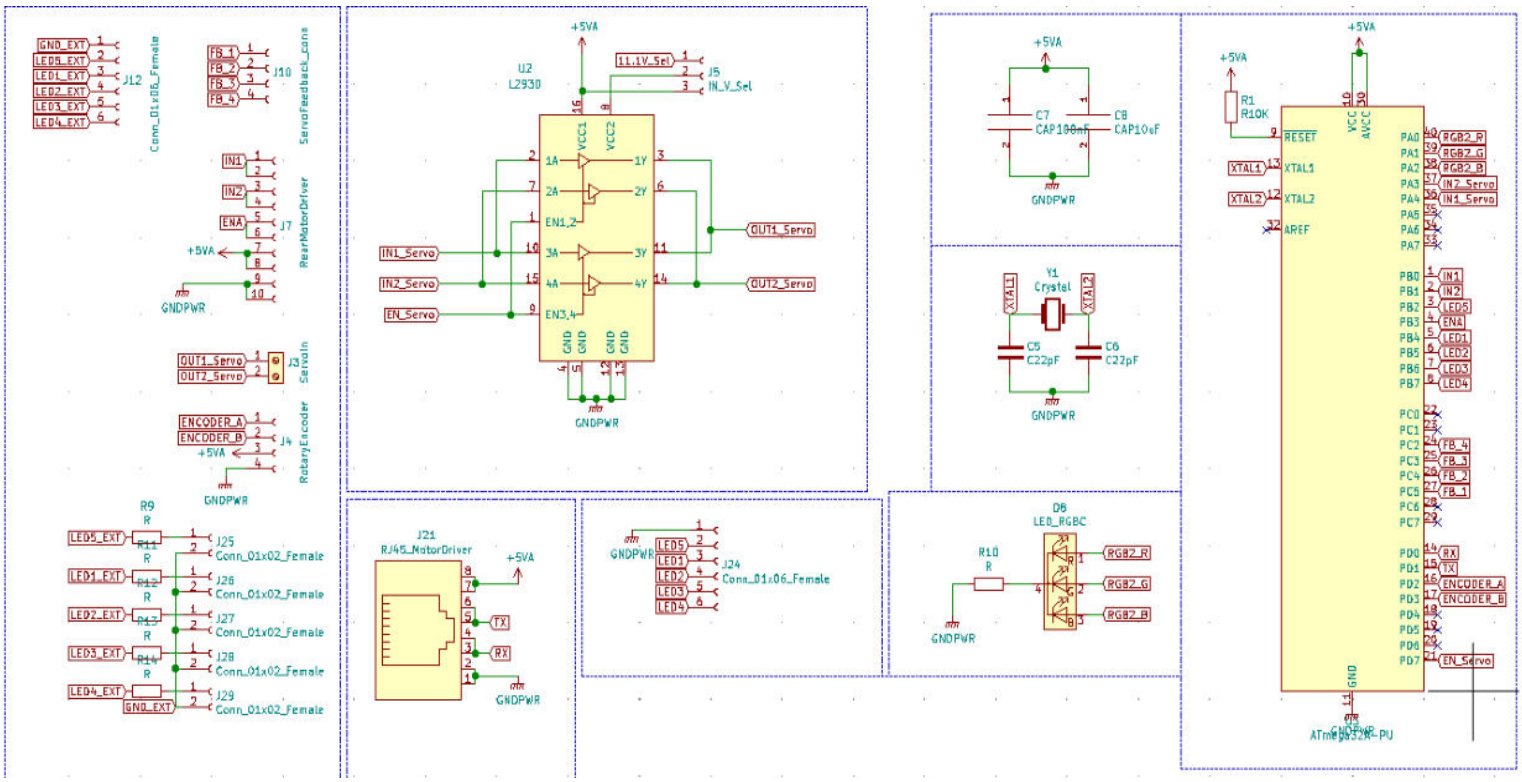
1- On Board 1A Motor Driver Circuit using L293D motor driver with internal Back EMF elimination (Using the 2 channels in parallel to increase power output).

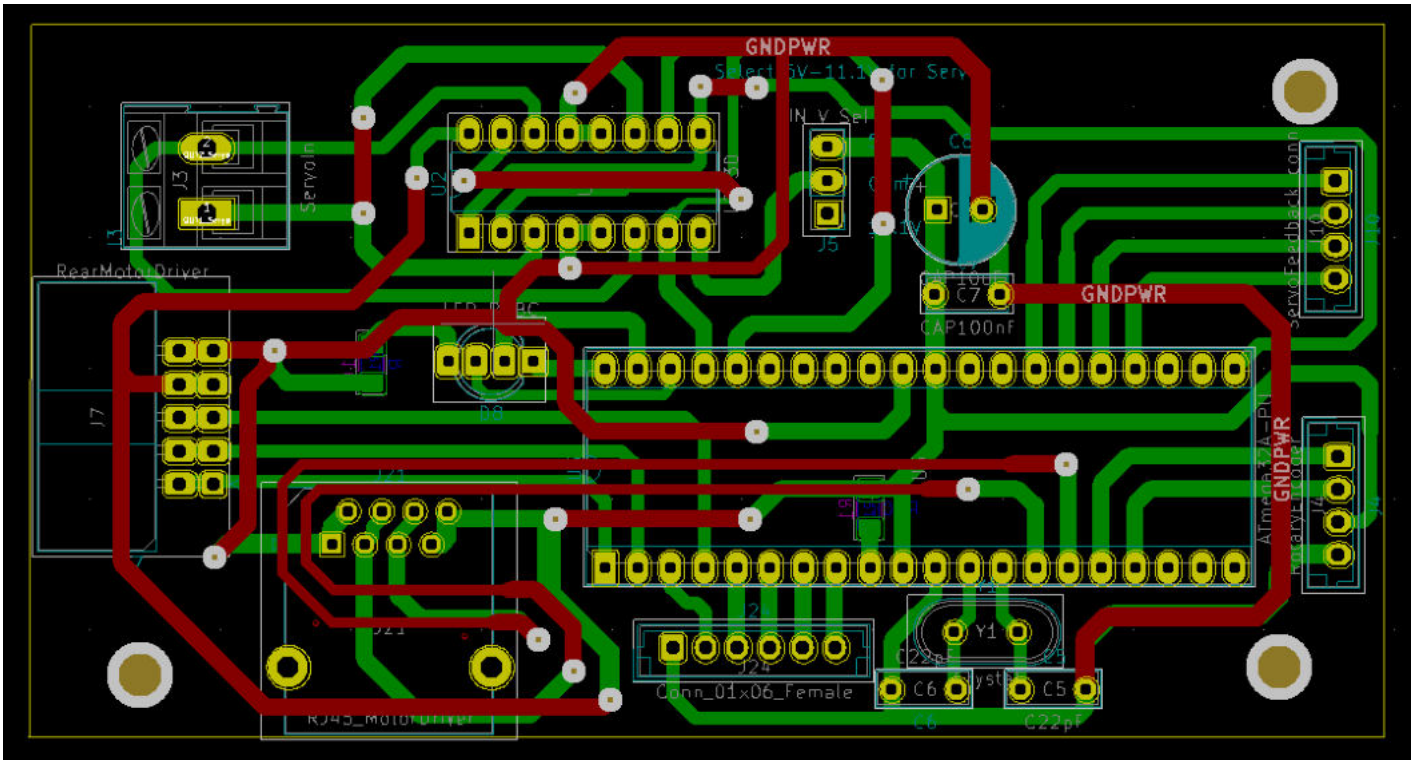
2- Front And Back LEDs control

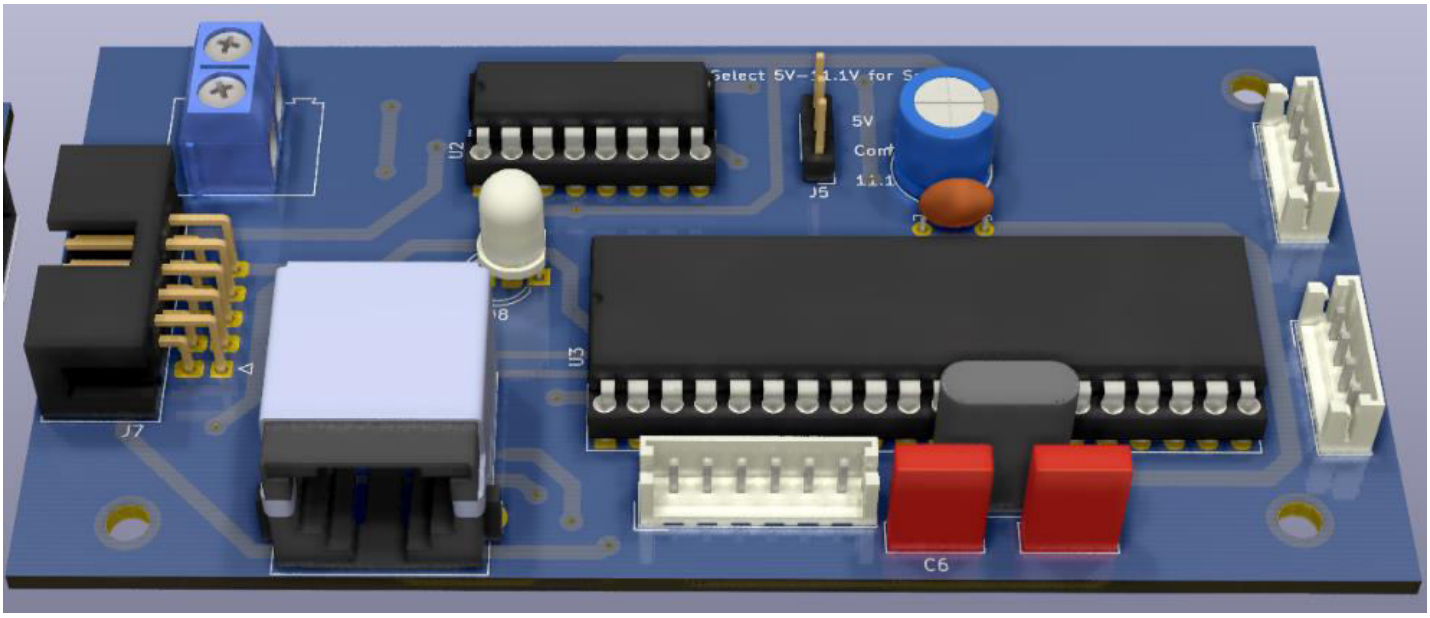
3- ATMEGA32A AVR RISC microcontroller

4- Selectable 2 Mode voltage selection for L293D (5V, 11.1V)

5- Ready to connect Quadrature Encoder or Rotary encoder to calculate speed for Future PID Implementation of Cruise control.




GNDPWR
GNDPWR
GNDPWR
RearMotorDriver
ServoIn
J3
J7
U2
J5
C7
CAP100nF
R8
J24
Conn_01x06_Female
Y1
C6
C5
C22pF
ServoFeedbackConn
RotaryFloorP

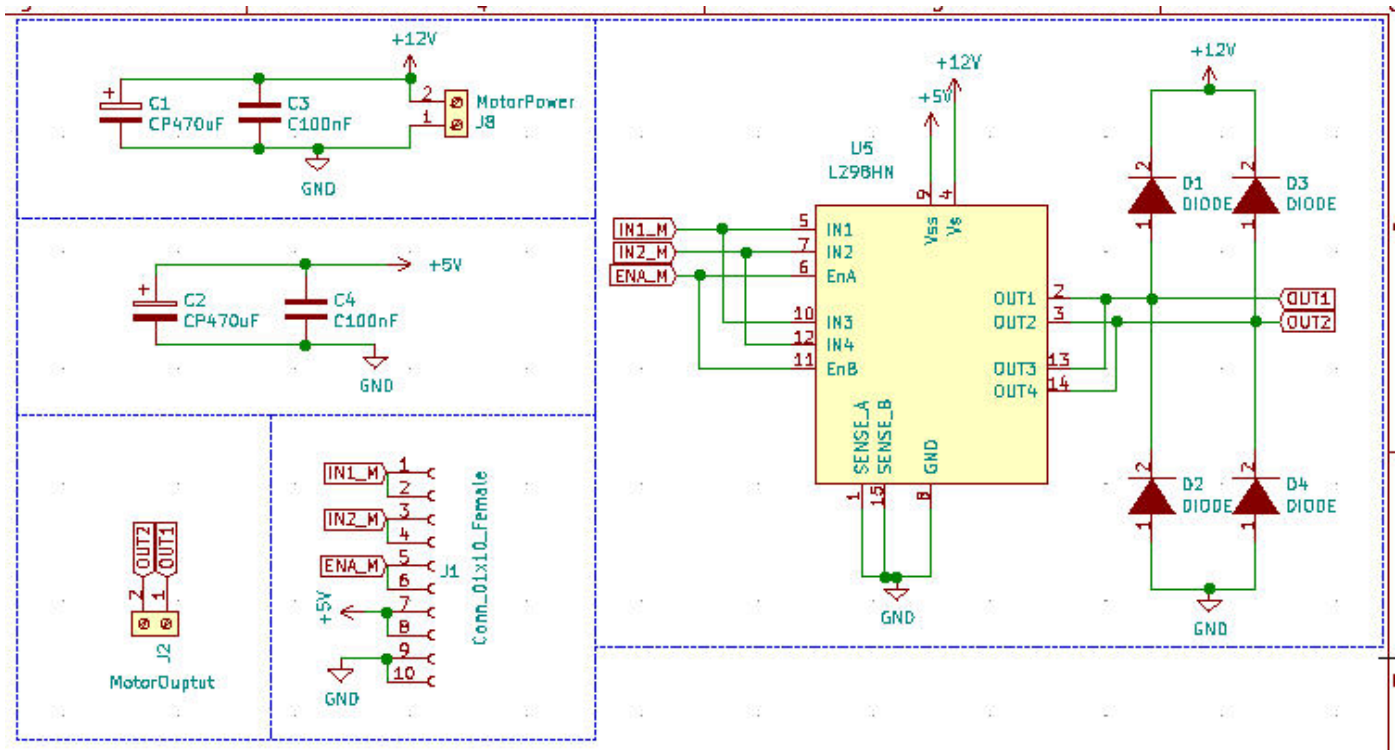

U2
J5
5V
J7
U3
C6

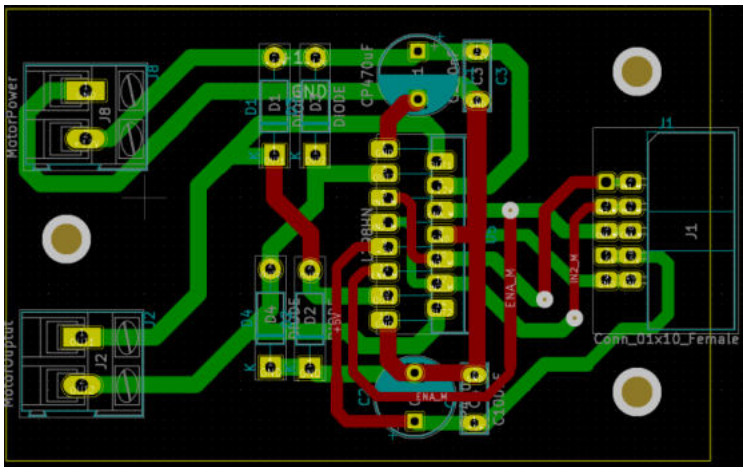
8.3.3 Motor Driver Board

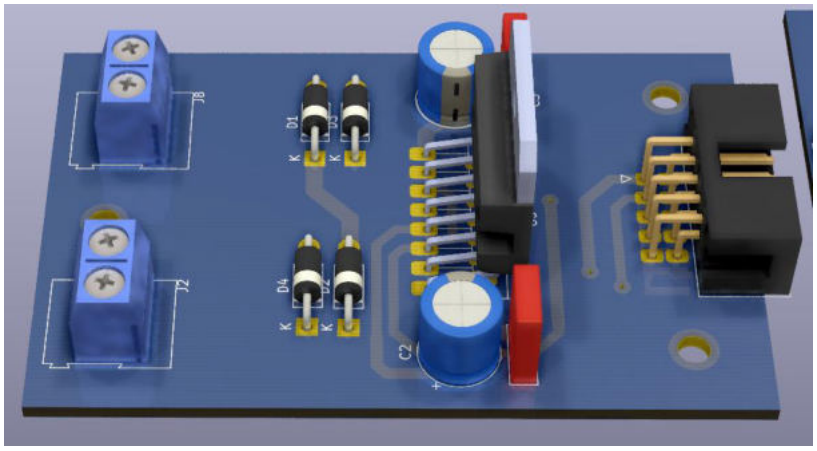
Features:

L298 Offer current output up to 3A per IC, by using the 2 channels in parallel.

Back EMF Elimination using the Didoes bridge

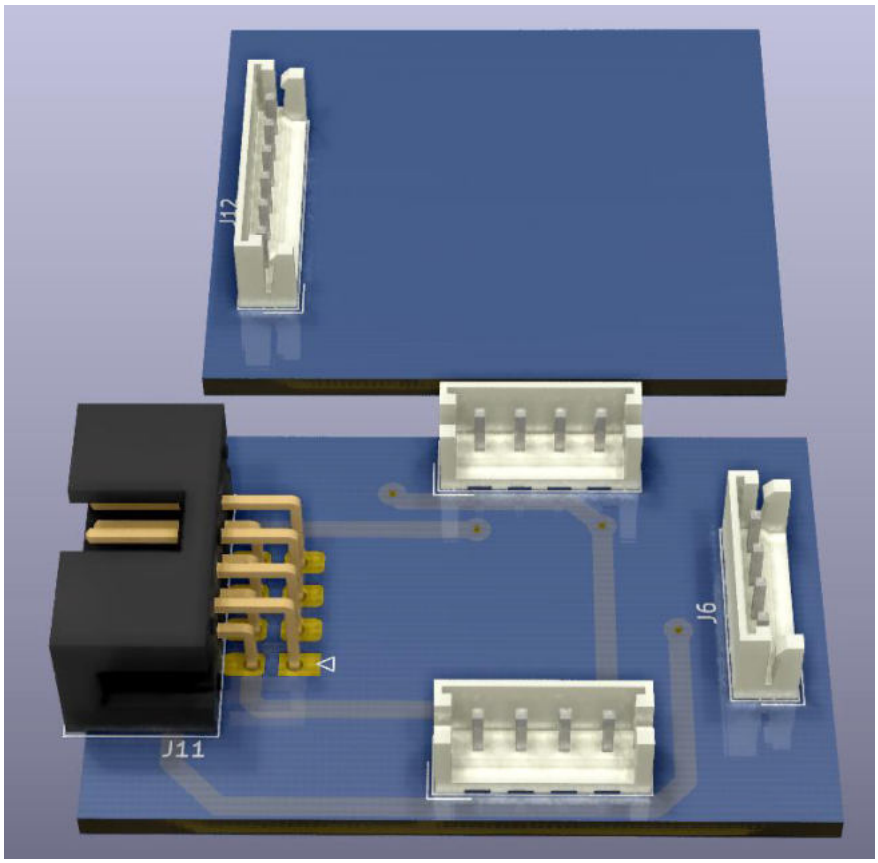






8.3.4 Extension Boards

We implemented to small extension board to help us in the wiring and troubleshooting process.

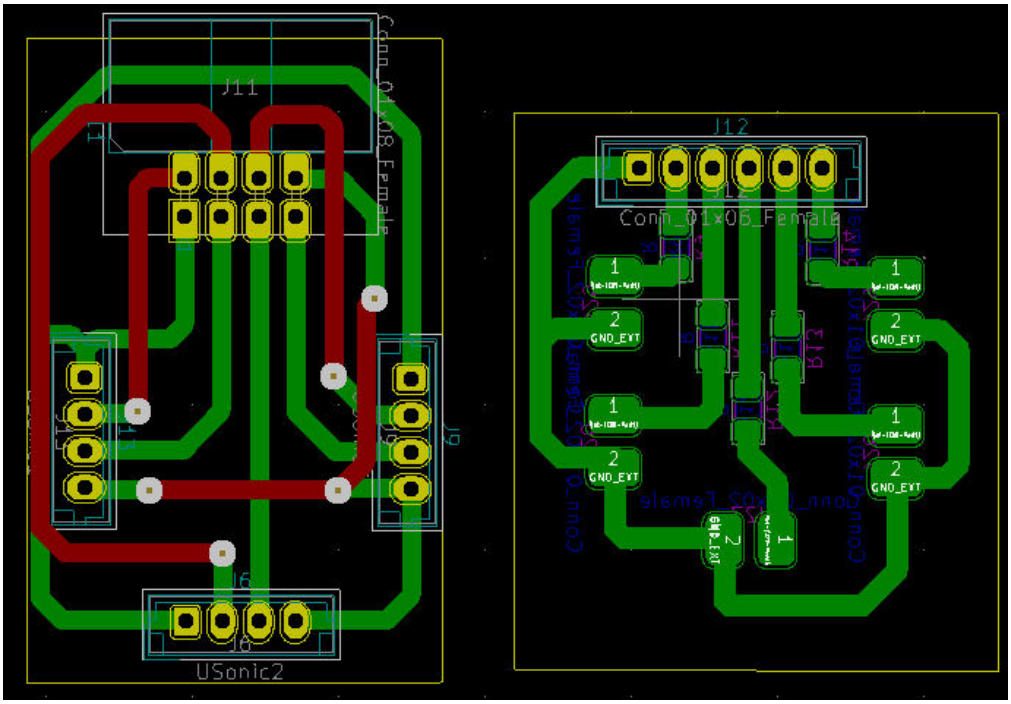


Ultrasonic Board

The 3 sensors collected into single socket for easy mounting and connection with the main board.

LEDs Board

For easy solder and wiring the LEDs Front and back into this small board at the bottom of the car as distribution hub



8.4 Sensors Used:

8.4.1 MPU6050

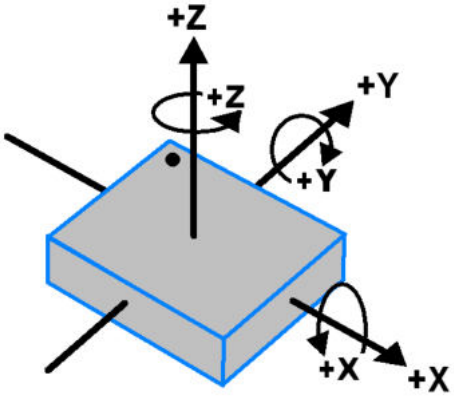
The MPU6050 consist of 3-axis Gyroscope & 3-axis Accelerometer with Micro Electro Mechanical System (MEMS) technology.

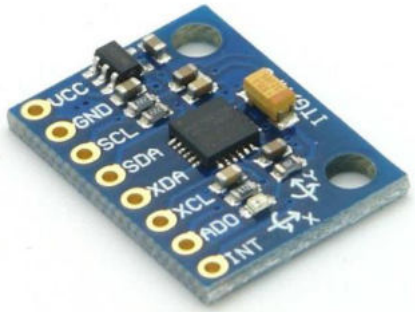
Accelerometer

Measure the acceleration in X,Y and Z directions by integrating the accelerometer readings we could get the car velocity in the 3 coordinates .

Gyroscope:

. It is used to detect rotational velocity along the X, Y, Z axes as shown in below figure.





8.4.2 Ultrasonic Sensor HC-SR04

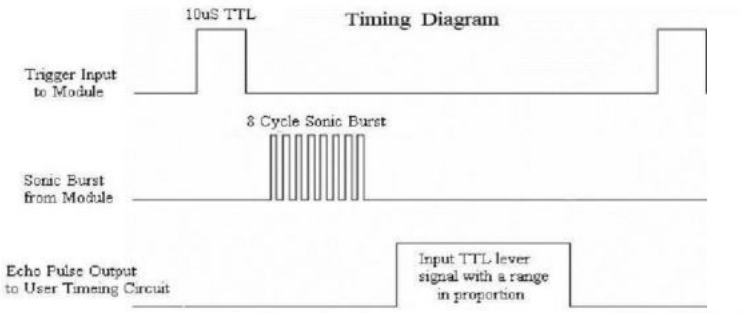
HC-SR04 provides 2cm - 400cm measurement range with ranging accuracy can reach to 3mm. The module includes ultrasonic transmitters, receiver and control circuit.

How it works

1- By applying pulse for at least 10us at TRIG pin.
2- The Module automatically sends eight 40 kHz and detect whether there is a pulse signal bouncing back.

3- IF the signal back, the echo pin fall to 0V , by calculating the time from high to low transition we get double the wave traveling time .

4- distance = (high level time ×velocity of sound (340M/S) / 2

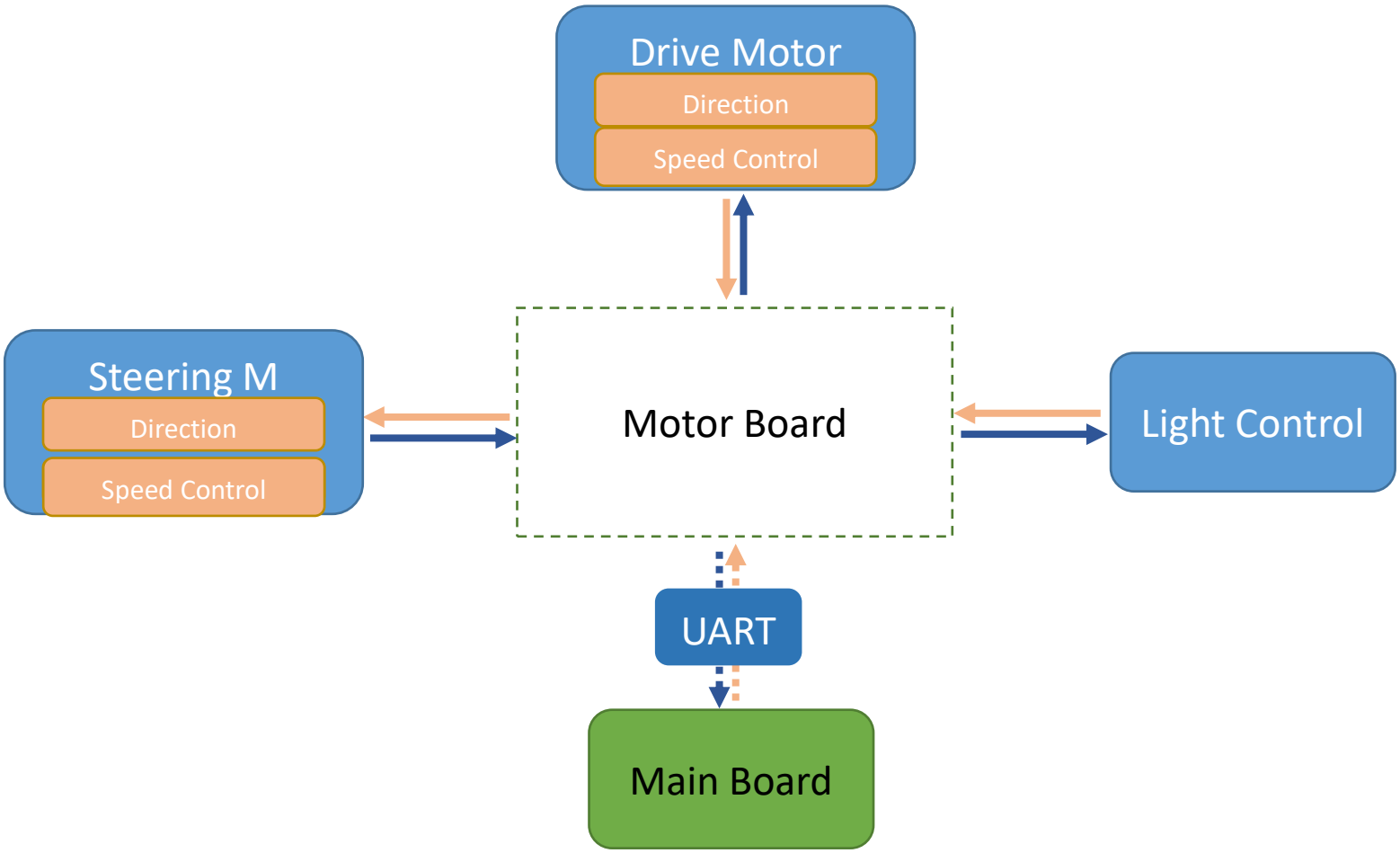


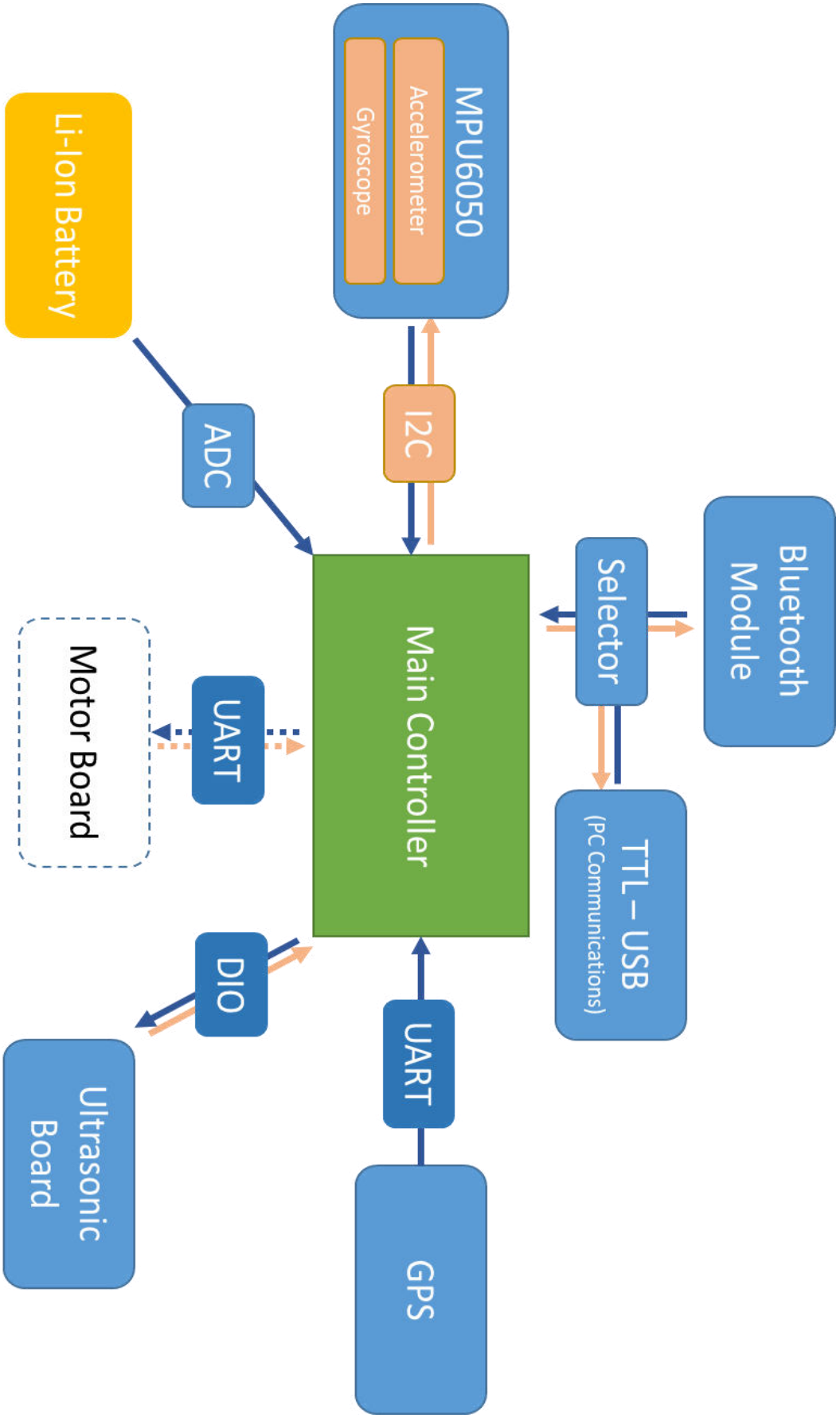
8.4.3 GPS Receiver Module (SKM53)

Low Power GPS receiver that uses UART protocol to broadcast Longitude and Latitude Data every 1-3 Seconds period.

Control the Front and Back motor in addition to lights.

8.5 ECUs Function Block Diagram




Bluetooth Module
Selector
TTL – USB
(PC Communications)
MPU6050
Accelerometer
Gyroscope
I2C
Main Controller
UART
GPS
ADC
Li-Ion Battery
UART
Motor Board
DIO
Ultrasonic Board

Conclusion & Future Work

- Self-driving cars could eliminate almost all road related accidents and fatalities. While this would be an amazing boon for humanity, they could do much else besides.

- The car depends on image processing theory and computer vision to take a stream of pictures and make decisions according to the picture which clarify the state of the road.

- In the future we hope to make localization that meaning that you can control the car to move from a specific point (A) to another point (B) and taking the shortest road.

- But we can't complete this feature because several problems such as its higher cost and there was no enough time, so for who interested in this field you can start from here

https://www.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

References

1- Machine Learning Refined Foundations, Algorithms and Applications Book by [JERMY WATT, REZA BORHANI AND AGGELOS K. KATSAGGELOS].

2- Control of Mobile Robot Course from edx website.

3- Plaskonka2015 Article Different Kinematic Path Follow in **[Paper].**

4- Hellstrom - Forward Kinematics**[Paper].**

5- https://www.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

6- http://docs.opencv.org

7- https://www.raspberrypi.org

8- https://www.numpy.org/

9- https://matplotlib.org/

10- https://www.tensorflow.org/

11- https://keras.io/

12- https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf **[Paper]**

13- https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf **[Paper]**

14- AUTOSAR Standard: https://www.autosar.org

15- Let Us C "5th edition": Yashava

Appendix A

Code

```c
#include "../SERVICES/STD_TYPES.h"
#include "../SERVICES/BIT_MATH.h"

#include "../MCAL/DIO/DIO_interface.h"
#include "../MCAL/PWM0/PWM0_interface.h"
#include "../MCAL/PWM2/PWM2_interface.h"
#include "../HAL/MOTORS/MOTORS_interface.h"
#include "../MCAL/USART/USART_interface.h"
#include "../HAL/LED/LED_interface.h"
#include "../HAL/RGB/RGB_interface.h"
#include "COMMANDS/COMMANDS_interface.h"

#include <util/delay.h>

int main(void) {

    USART_voidInit();
    PWM0_voidInit();
    PWM2_voidInit();
    MOTORS_voidInit();
    LED_voidInit();
    RGB_voidInit();

    RGB_Blinking();

    u8 * DataStringRx = USART_u8ptrReceivedString();
    u8 x = 0;
    while (1) {
        //LED_voidON_ONE(LED_FRONT_L);
        DataStringRx = USART_u8ptrReceivedString();
        x = COMMANDS_u8Check((u8 *) DataStringRx, 20);

    }
    return 0;
```

```c
#include "../../SERVICES/BIT_MATH.h"
#include "../../SERVICES/STD_TYPES.h"

#include "../../MCAL/DIO/DIO_interface.h"
#include "../../MCAL/PWM0/PWM0_interface.h"
#include "../../MCAL/PWM2/PWM2_interface.h"
#include "../../MCAL/USART/USART_interface.h"
#include "../../HAL/MOTORS/MOTORS_interface.h"
#include "../../HAL/LED/LED_interface.h"

#include "COMMANDS_interface.h"

u8 Return_Command = '0';
u8 Speed = 0;
u8 Direction = MOTOR_STOP;

u8 COMMANDS_u8Check(u8 * Copy_u8ptrCommandStringRx, u8 Copy_u8Speed) {

    if (Copy_u8ptrCommandStringRx[0] == 'M') {

        /*Check Back MOTOR*/
        if (Copy_u8ptrCommandStringRx[1] == '2') {

            /*Motors Speed Commands*/
            switch (Copy_u8ptrCommandStringRx[2]) {
            case 'V':
                    /*Set Speed */
                    Speed = Copy_u8ptrCommandStringRx[3];
                    USART_voidTransmitterString((u8 *) "M2V-OK\n");
                    MOTOR_BACK_voidDirection_Speed(Direction, Speed);
                    Return_Command = '9';
                    return Return_Command;
                break;
```

```
        /*Motors Directions Commands*/
    case 'F':
        USART_voidTransmitterString((u8 *) "M2F-OK\n");
        MOTOR_BACK_voidDirection_Speed(MOTOR_FROWARD, Copy_u8Speed);
        Direction = MOTOR_FROWARD;
        Return_Command = '5';
        return Return_Command;
    case 'B':
        USART_voidTransmitterString((u8 *) "M2B-OK\n");
        MOTOR_BACK_voidDirection_Speed(MOTOR_BACK, Copy_u8Speed);
        Direction = MOTOR_BACK;
        Return_Command = '6';
        return Return_Command;
    case 'S':
        USART_voidTransmitterString((u8 *) "M2S-OK\n");
        MOTOR_BACK_voidDirection_Speed(MOTOR_STOP, 0);
        Return_Command = '7';
        return Return_Command;
    case 'X':
        USART_voidTransmitterString((u8 *) "M2X-OK\n");
        MOTOR_BACK_voidDirection_Speed(MOTOR_BREAK, 255);
        Return_Command = '8';
        return Return_Command;
    default:
        USART_voidTransmitterString((u8 *) "NOT-OK\n");
        Return_Command = '0';
        return Return_Command;
    }

}

/*Check Front MOTOR*/
else if (Copy_u8ptrCommandStringRx[1] == '1') {
    switch (Copy_u8ptrCommandStringRx[2]) {
```

```
case 'R':
    USART_voidTransmitterString((u8 *) "M1R-OK\n");
    MOTOR_FRONT_voidDirection_Speed(MOTOR_RIGHT, Copy_u8Speed);
    Return_Command = '1';
    return Return_Command;

case 'L':
    USART_voidTransmitterString((u8 *) "M1L-OK\n");
    MOTOR_FRONT_voidDirection_Speed(MOTOR_LEFT, Copy_u8Speed);
    Return_Command = '2';
    return Return_Command;

case 'S':
    USART_voidTransmitterString((u8 *) "M1S-OK\n");
    MOTOR_FRONT_voidDirection_Speed(MOTOR_STOP, 0);
    Return_Command = '3';
    return Return_Command;

case 'X':
    USART_voidTransmitterString((u8 *) "M1X-OK\n");
    MOTOR_FRONT_voidDirection_Speed(MOTOR_BREAK, 255);
    Return_Command = '4';
    return Return_Command;

case 'D':
    if (Copy_u8ptrCommandStringRx[3] == '?') {
        switch (Return_Command) {
        case '1':
            USART_voidTransmitterString((u8 *) "M1D?-R\n");
            break;
        case '2':
            USART_voidTransmitterString((u8 *) "M1D?-L\n");
            break;
        case '3':
            USART_voidTransmitterString((u8 *) "M1D?-S\n");
```

```
        case 'L':
            LED_voidON_ONE(LED_FRONT_L);
            Return_Command = '0';
            return Return_Command;
            break;
        default:
            USART_voidTransmitterString((u8 *) "NOT-OK\n");
            Return_Command = '0';
            return Return_Command;
            break;
        }
    } else if (Copy_u8ptrCommandStringRx[3] == 'F') {
        switch (Copy_u8ptrCommandStringRx[2]) {
        case 'R':
            LED_voidOFF_ONE(LED_FRONT_R);
            Return_Command = '0';
            return Return_Command;
            break;
        case 'L':
            LED_voidOFF_ONE(LED_FRONT_L);
            Return_Command = '0';
            return Return_Command;
            break;
        default:
            USART_voidTransmitterString((u8 *) "NOT-OK\n");
            Return_Command = '0';
            return Return_Command;
            break;
        }
    }
}

else if (Copy_u8ptrCommandStringRx[1] == 'B') {
    if (Copy_u8ptrCommandStringRx[3] == 'O') {
        switch (Copy_u8ptrCommandStringRx[2]) {
```

```
else if (Copy_u8ptrCommandStringRx[1] == 'B') {
    if (Copy_u8ptrCommandStringRx[3] == 'O') {
        switch (Copy_u8ptrCommandStringRx[2]) {
        case 'R':
            LED_voidON_ONE(LED_BACK_R);
            Return_Command = '0';
            return Return_Command;
            break;
        case 'L':
            LED_voidON_ONE(LED_BACK_L);
            Return_Command = '0';
            return Return_Command;
            break;
        default:
            USART_voidTransmitterString((u8 *) "NOT-OK\n");
            Return_Command = '0';
            return Return_Command;
            break;
        }
    } else if (Copy_u8ptrCommandStringRx[3] == 'F') {
        switch (Copy_u8ptrCommandStringRx[2]) {
        case 'R':
            LED_voidOFF_ONE(LED_BACK_R);
            Return_Command = '0';
            return Return_Command;
            break;
        case 'L':
            LED_voidOFF_ONE(LED_BACK_L);
            Return_Command = '0';
            return Return_Command;
            break;
        default:
            USART_voidTransmitterString((u8 *) "NOT-OK\n");
            Return_Command = '0';
            return Return_Command;
```

```
                default:
                    USART_voidTransmitterString((u8 *) "NOT-OK\n");
                    Return_Command = '0';
                    return Return_Command;
                    break;
                }
            }
        }

        else {
            USART_voidTransmitterString((u8 *) "NOT-OK\n");
            Return_Command = '0';
            return Return_Command;
        }
    } else {
        USART_voidTransmitterString((u8 *) "NOT-OK\n");
        Return_Command = '0';
        return Return_Command;
    }
    USART_voidTransmitterString((u8 *) "NOT-OK\n");
    Return_Command = '0';
    return Return_Command;
}
```