**PATH TRACKING USING MODEL PREDICTIVE CONTROLS AND INTERIOR POINT OPTIMIZATION**

## A PROJECT REPORT

## SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE

## OF

# BACHELOR OF TECHNOLOGY

# IN

# **MATHEMATICS AND COMPUTING ENGINEERING**

# Submitted by:

# **SAMKIT JAIN (DTU/2K14/MC/068)**

# **SARANSH AGARWAL (DTU/2K14/MC/071)**

# **ROHAN RAJ (DTU/2K14/MC/063)**

Under the supervision of:

# Prof. Sangita Kansal

# Prof. Anjana Gupta

# Dr. Naokant Deo



**DEPARTMENT OF APPLIED MATHEMATICS**

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Bawana Road, Delhi - 110042

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Bawana Road, Delhi - 110042

**CANDIDATE’S DECLARATION**

We, Samkit Jain (DTU/2K14/MC/068), Saransh Agarwal (DTU/2K14/MC/071), Rohan Raj (DTU/2K14/MC/063), students of B.Tech (Mathematics and Computing Engineering), hereby declare that the project Dissertation titled “Path tracking using Model Predictive Controls and Interior point optimization” which is submitted by us to the Department of Applied Mathematics, Delhi Technological University, Delhi in partial fulfillment of the requirement for the award of the degree of Bachelor of Technology, is original and not copied from any source without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma Associateship, Fellowship or other similar title or recognition.

Place: Delhi **Samkit Jain**

Date: **DTU/2K14/MC/068**

**Saransh Agarwal**

**DTU/2K14/MC/071**

**Rohan Raj**

**DTU/2K14/MC/063**

**DEPARTMENT OF APPLIED MATHEMATICS**

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Bawana Road, Delhi - 110042

**Certificate**

I hereby certify that the Project Dissertation titled “Path tracking using Model Predictive Control and Interior Point Optimization” which is submitted by Samkit Jain (DTU/2K14/MC/068), Saransh Agarwal (DTU/2K14/MC/071) and Rohan Raj (DTU/2K14/MC/063), students of B.Tech (Mathematics and Computing Engineering), Delhi Technological University, Delhi in partial fulfillment of the requirement for the award of degree of Bachelor of Technology, is a record of the project work carried out by the students under my supervision. To the best of my knowledge this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

Place: Delhi **Prof. Sangita Kansal**

Date: HOD, Professor

Department of Applied Mathematics

**Prof. Anjana Gupta**

Professor

Department of Applied Mathematics

**Dr. Naokant Deo**

Associate Professor

Department of Applied Mathematics

# ACKNOWLEDGEMENTS

# 

We owe a debt of deepest gratitude to my mentor Prof. Sangeeta Kansal , Prof. Anjana Gupta and Dr. Naokant Deo (Associate Professor) for their guidance, support, motivation and encouragement throughout the period this work was carried out. Their readiness for consultation at all times, their educative comments, concern and assistance even with practical things have been invaluable. We also thank other faculty and staff members of our department for their invaluable help and guidance.

# ABSTRACT

In this project, we have attempted to implement the human interpretation of driving task, by making an algorithm to understand a path, and get an idea on how to direct the car on that path. A path made up of circular markings is identified using a camera mounted on the car, and then based on a polynomial curve generated by fitting the circles on the contours, and using the centres of the circles. The work is done in the car frame. Based on the reference path, an optimization problem is drafted, which minimises the deviation from the desired path, subject to some actuator constraints. This non-linear convex optimization problem is solved using interior point optimization.

# LIST OF FIGURES

# 

2.1 Top view after perspective transformation

2.2 Perspective transformation diagram

2.3 Image thresholding example

2.4 Thresholding on Saturation channel

2.5 Erosion example

2.6 Dilation example

2.7 Contouring on a single frame of track

2.8 RGB color model

2.9 HSV hexacone

2.10 HLS double cone

2.11 H color channel from HSV

2.12 S color channel from HSV

2.13 Circles identified and marked

2.14 Polynomial fitted on the centers of the circles

3.1 Basic structure of MPC

3.2 Kinematic Bicycle Model

3.3 Steering Constraint

3.4 Orientation error

3.5 Cross Track Error

3.6 Simulation Results

4.1 GPIO pinout diagram and Raspberry Pi Model 3B

4.2 Pulse Width Modulation

4.3 H-Bridge Circuit

# CONTENTS

# 

**Candidate’s Declaration i**

**Certificate ii**

**Acknowledgement iii**

**Abstract iv**

**Contents v**

**List of Figures vi**

**CHAPTER - 1 INTRODUCTION 8**

1.1 Potential Benefits 8

1.2 Potential Problems 9

1.3 Industry and Future 9

**CHAPTER - 2 COMPUTER VISION 10**

2.1 Perspective Transform 10

2.1.1 Frames of Reference 11

2.1.2 Examples of Useful Code 12

2.2 Image Thresholding 12

2.3 Morphological Transformations 13

2.3.1 Erosion 13

2.3.2 Dilation 14

2.4 Contouring 15

2.5 Minimum Enclosing Circle 15

2.6 Moments 16

2.7 Color Spaces 16

2.7.1 RGB Color Space 16

2.7.2 HSV Color Space 18

2.7.2 HLS Color Space 19

2.8 Our results on simulations 20

**CHAPTER - 3 MODEL PREDICTIVE CONTROL 25**

3.1 Introduction 25

3.1.1 Idea behind control theory 25

3.1.2 MPC (Model Predictive Control) 25

3.2 Overview of MPC 26

3.3 Theory Behind MPC 26

3.3.1 Principles of MPC 27

3.4 Vehicle Model 27

3.5 Constraints 29

3.6 Reference State 30

3.7 Optimization 33

3.8 Simulation Results 33

**CHAPTER - 4 HARDWARE 46**

4.1 Raspberry Pi 46

4.2 Brushed DC Motors 47

4.2.1 Pulse Width Modulation 48

4.3 H-Bridge Circuit49

**REFERENCES 52**

# CHAPTER - 1

# INTRODUCTION

Autonomous vehicles have vast potential applications in a wide spectrum of domains such

as transport, exploration, search and rescue, environmental management, and reconnaissance.

Building a fully autonomous vehicle that can reliably navigate and avoid obstacles at high speed

is a major challenge for robotics, and a new domain of application for machine learning research.

The groundwork for this project was done in a Defense Advanced Research Projects Agency (DARPA) seedling project known as DARPA Autonomous Vehicle (DAVE) in which a sub-scale radio control (RC) car drove through a junk-filled alleyway. DAVE was trained on hours of human driving in similar, but not identical environments.

1.1 Potential Benefits

Autonomous vehicles bring plenty of potential benefits. While driving conventional vehicles causes a rise of a cortisol level, known as a stress hormone, riders in AVs won’t experience the stress level related to driving and they could instead rest or work while traveling. Elimination of the taxi and truck drivers need will reduce the price for some of the services. Autonomous vehicles could be used by a non-drivers and will enable them to freely use cars without any dependence on other people. Increased safety will reduce many common accident risks and also related crash cost and insurance payments. Increased road capacity and reduced cost by supporting platooning of vehicles that are able to drive in close distance, what further means saving a fuel due to a decreased resistance of vehicles. Autonomous vehicle will offer more efficient parking when dropping off a passenger, the car can find it’s parking spot by itself. What can reduce the waiting time for passenger and reduce the parking price, since the car can park itself in a cheaper area. AVs can reduce operational CO2 emissions of vehicles and increase fuel efficiency, due to a fact that they will drive more optimally than a human driver. If AVs will be used as a shared resource new car sharing services can provide several savings.

1.2 Potential Problems

AVs can increase costs related to additional car equipment, services, and further maintenance, and further investments in roadway infrastructure will be also necessary. AVs may introduce new risks, in a sense of system failures that can occur. What can mean that AVs could be less safe in certain situations and conditions. Being connected to a cloud and operated by a central unit system, there will be security and privacy concerns related to cyber security threats, where vehicles can be controlled remotely. Further vulnerable abuse of information, tracking and data sharing could violate the passenger privacy and those cars could be used for some terrorist activities.

1.3 Industry and Future

There are many forces that drive the innovation and autonomous vehicles. Until nowadays, a car manufacturers were the leaders in the automotive industry, by holding most of the competitive power over the other stakeholders and suppliers. Now, with a new emerging technology; multiple new stakeholders are coming into a battle over the future automotive and transportation market. Companies such as Google, Uber, and Tesla are not the typical car manufacturers, yet they are the top developers of the autonomous technology right now. The interplay between those tech developers, car manufactures and ride-hailing service providers will redistribute the power and roles in the automotive market as we know it nowadays.

# CHAPTER - 2

# COMPUTER VISION

2.1 Perspective transform

When human eyes see near things they look bigger as compare to those who are far away. This is called perspective in a general way. Whereas transformation is the transfer of an object e.t.c from one state to another.

So overall, the perspective transformation deals with the conversion of 3d world into 2d image. The same principle on which human vision works and the same principle on which the camera works.

A perspective transform maps the points in a given image to different, desired, image points with a new perspective. The perspective transform we’ll be most interested in is a bird’s-eye view transform that let’s us view a scene from above; this will be useful for calculating the lane curvature later on. Aside from creating a bird’s eye view representation of an image, a perspective transform can also be used for all kinds of different view points.

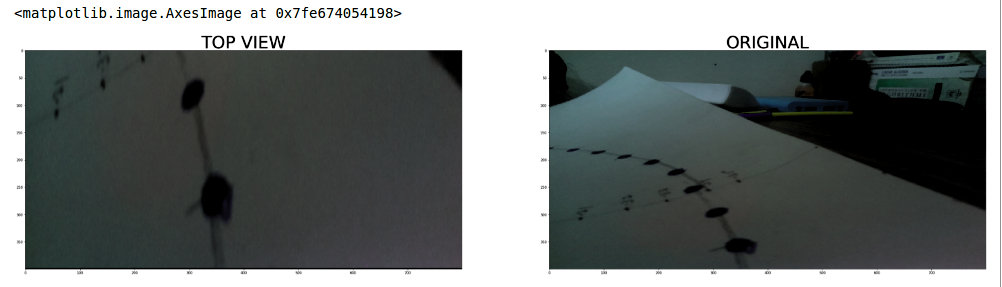


Figure 2.1

2.1.1 Frames of reference

In order to analyze a 3d world/image/scene, 5 different frame of references are required.

* Object
* World
* Camera
* Image
* Pixel

### **Object coordinate frame**

Object coordinate frame is used for modeling objects. For example, checking if a particular object is in a proper place with respect to the other object. It is a 3d coordinate system.

### **World coordinate frame**

World coordinate frame is used for co-relating objects in a 3 dimensional world. It is a 3d coordinate system.

### **Camera coordinate frame**

Camera co-ordinate frame is used to relate objects with respect of the camera. It is a 3d coordinate system.

### **Image coordinate frame**

It is not a 3d coordinate system, rather it is a 2d system. It is used to describe how 3d points are mapped in a 2d image plane.

### **Pixel coordinate frame**

It is also a 2d coordinate system. Each pixel has a value of pixel co ordinates.

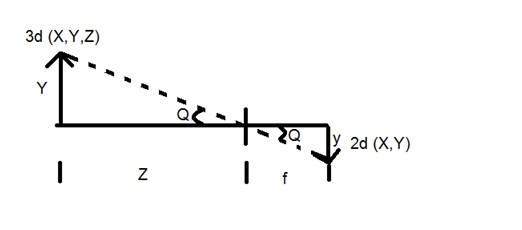


Figure 2.2

Source: https://www.tutorialspoint.com/dip/perspective\_transformation.htm

2.1.2 Examples of Useful Code

Compute the perspective transform, M, given source and destination points:

M = cv2.getPerspectiveTransform(src, dst)

Warp an image using the perspective transform, M:

warped = cv2.warpPerspective(img, M, img\_size, flags=cv2.INTER\_LINEAR)

2.2 Image thresholding

Image thresholding is a simple, yet effective, way of partitioning an image into a foreground and background. If pixel value is greater than a threshold value, it is assigned one value (may be white), else it is assigned another value (may be black). The function used is **cv2.threshold**.

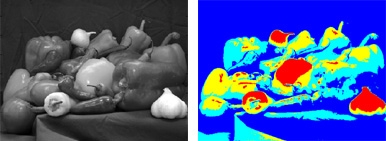


Figure 2.3

For this project, we have first computed the top view of the road using perspective transformation. Then we have smoothed the image using Gaussian Blur. For blurring the image we chose a kernel of size (3,3). This was a tunable parameter and a kernel of size (3,3) produced the best results.

After this, we converted the RGB image to an HSV image and applied thresholds to the Hue and Saturation channels of the frame. The results are as shown in Figure 2.4.

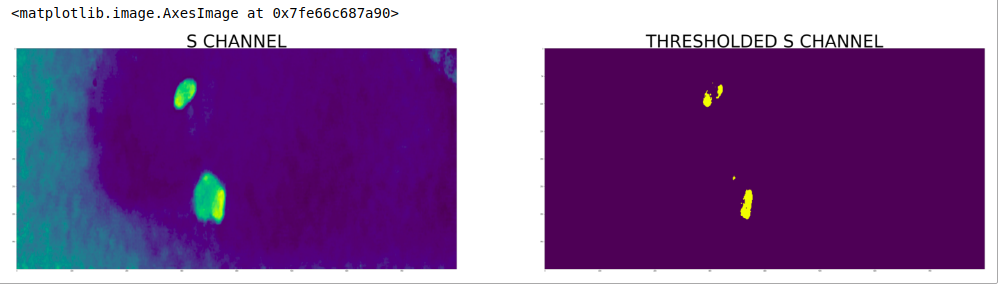


Figure 2.4

2.3 Morphological Transformations

Morphological transformations are some simple operations based on the image shape. It is normally performed on binary images. It needs two inputs, one is our original image, second one is called structuring element or kernel which decides the nature of operation. Two basic morphological operators are Erosion and Dilation. Then its variant forms like Opening, Closing, Gradient etc also comes into play.

2.3.1 Erosion

The basic idea of erosion is just like soil erosion only, it erodes away the boundaries of foreground object (Always try to keep foreground in white). The kernel slides through the image (as in 2D convolution). A pixel in the original image (either 1 or 0) will be considered 1 only if all the pixels under the kernel is 1, otherwise it is eroded (made to zero).

So what happens is that, all the pixels near boundary will be discarded depending upon the size of kernel. So the thickness or size of the foreground object decreases or simply white region decreases in the image. It is useful for removing small white noises (as we have seen in colorspace chapter), detach two connected objects etc.

Figure 2.5

Source: https://docs.opencv.org/trunk/d9/d61/tutorial\_py\_morphological\_ops.html

In Figure 2.5, the image on the left is the original image and the one on the right is the eroded image.

2.3.2 Dilation

It is just opposite of erosion. Here, a pixel element is '1' if atleast one pixel under the kernel is '1'. So it increases the white region in the image or size of foreground object increases. Normally, in cases like noise removal, erosion is followed by dilation. Because, erosion removes white noises, but it also shrinks our object. So we dilate it. Since noise is gone, they won't come back, but our object area increases. It is also useful in joining broken parts of an object.

Figure 2.6

Source: https://docs.opencv.org/trunk/d9/d61/tutorial\_py\_morphological\_ops.html

In Figure 2.6, the image on the left is the original image and the one on the right is the dilated image.

2.4 Contouring

Contours can be explained simply as a curve joining all the continuous points (along the boundary), having same color or intensity. The contours are a useful tool for shape analysis and object detection and recognition.

* For better accuracy, use binary images. So before finding contours, apply threshold or canny edge detection.
* In OpenCV, finding contours is like finding white object from black background. So remember, object to be found should be white and background should be black.

For this project, we use the openCV method findContours() to find contours in our frame and then use that to compute the circumcircle of our contours.



Figure 2.7

Results of contouring on one frame is shown in Figure 2.7.

2.5 Minimum Enclosing Circle

Next we find the circumcircle of an object using the function **cv2.minEnclosingCircle()**. It is a circle which completely covers the object with minimum area. The function finds the minimal enclosing circle of a 2D point set using an iterative algorithm.

2.6 Moments

In image processing, computer vision and related fields, an **image moment** is a certain particular weighted average (moment) of the image pixels' intensities, or a function of such moments, usually chosen to have some attractive property or interpretation.

Image moments are useful to describe objects after segmentation. Simple properties of the image which are found *via* image moments include area (or total intensity), its centroid, and information about its orientation. Image moments help you to calculate some features like center of mass of the object, area of the object etc.

Image with pixel intensities *I*(*x*,*y*), raw image moments *Mij* are calculated by

 (2.1)

Centroids are given by

 (2.2)

 (2.3)

2.7 Color Spaces

In this project, we have worked with RGB, HSV, HLS and grayscale color schemes. We will discuss these briefly.

2.7.1 RGB Color Space

Based on the tristimulus theory of vision, our eyes perceive color through the stimulation of three visual pigments in the cones of the wtina. These visual pigments have a peak sensitivity at wavelengths of about 630 run (red), 530 nm (green), and 450 nm (blue). By comparing intensities in a light source, we perceive the color of the light. This theory of vision is the basis for displaying color output on a video monitor using the three color primaries, red, green, and blue, referred to as the RGB color model.

We can represent this model with the unit cube defined on R, C, and B axes,

as shown in Fig. 2.8. The origin represents black, and the vertex with coordinates (1, 1 , l ) is white. Vertices of the cube on the axes represent the primary colors, and the remaining vertices represent the complementary color for each of the primary colors. As with the XYZ color system, the RGB color scheme is an additive model. Intensities of the primary colors are added to produce other colors. Each color point within the bounds of the cube can be represented as the triple (R , G, B) where values for R, G, and B are assigned in the range from 0 to 1. Thus, a color

C, is expressed in RGB components as

 (2.4)

The magenta vertex is obtained by adding red and blue to produce the triple (1,0,l). and white at (1,l,1) is the sum of the red, green, and blue vertices. Shades of gray are represented along the main diagonal of the cube from the origin (black) to the white vertex. Each point along this diagonal has an equal contribution from each primary color, so that a gray shade halfway between black and white is represented as (0.5, 0.5, 0.5).

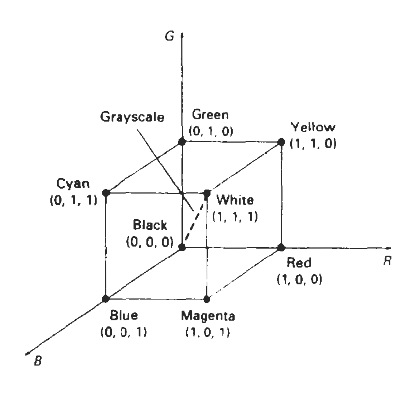


Figure 2.8

Source: Computer Graphics by Donald Hearn and M. Pauline Baker

2.7.2 HSV Color Space

lnstead of a set of color primaries, the HSV model uses color descriptions that have a more intuitive appeal to a user. To give a color specification, a user selects a spectral color and the amounts of white and black that are to be added to obtain different shades, tints, and tones. Color parameters in this model are hue ( H ), saturation ( S ), value ( V ).

Hue is represented as an angle about the vertical axis, ranging from 0 at red through 360. Vertices of the hexagon are separated by 60 intervals. Yellow is at 60, green at 120, and cyan opposite red at H = 180. Complementary colors are 180 apart

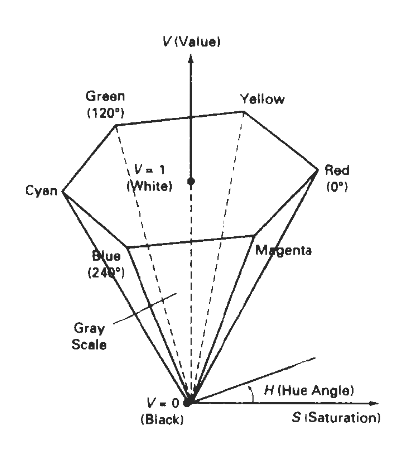


Figure 2.9

Source: Computer Graphics by Donald Hearn and M. Pauline Baker

2.7.3 HLS Color Model

Another model based on intuitive color parameters is the HLS system used by Tektronix. This model has the double-cone representation shown in Figure 2.10. The three color parameters in this model are called Hue (H), Lightness (L), and Saturation (S).

Hue has the same meaning as in the HSV model. It specifies an angle about the vertical axis that locates a chosen hue. In this model, H = 0' corresponds to blue. The remaining colors are specified around the perimeter of the cone in the same order as in the HSV model. Magenta is at 60, red is at 120, and cyan is located at H = 180. Again, complementary colors are 180 apart on the double cone.

The vertical axis in this model is called lightness, L. At L = 0, we have black, and whitc is at L = 1. Gray scale is along the L axis, and the "pure hues" lie on the L = 0.5 plane. Saturation parameter S again specifies relative purity of a color. This parameter varies from 0 to 1, and pure hues are those for which S = 1 and L = 0.5. As S decreases, the hues are said to be less pure. At S = 0, we have the gray scale.

As in the HSV model, the HLS system allows a user to think in terms of making a selected hue darker or lighter. A hue is selected with hue angle H, and the desired shade, tint, or tone is obtained by adjusting L and S. Colors are made lighter by increasing L and made darker by decreasing L. When S is decreased, the colors move toward gray.

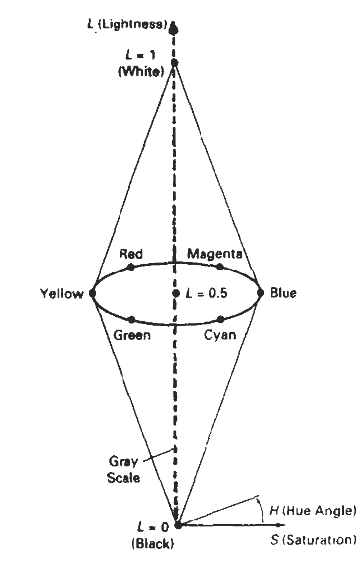


Figure 2.10

Source: Computer Graphics by Donald Hearn and M. Pauline Baker

2.8 Simulation attempt on our data

We used all the methods mentioned above, and created a script that would help, for a car with a camera mounted on the top, determine the circles and then fit a polynomial, on the path, so that we receive the reference path which can be used in the cross track error.

Code :

import cv2

import time

import numpy as np

class ComputerVision:

centroids = []

# Init functions

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

centroids = []

# Helper functions for color space change

def bgr\_to\_rgb(self,image):

return cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

def rgb\_to\_gray(self,image):

return cv2.cvtColor(image,cv2.COLOR\_RGB2GRAY)

def rgb\_to\_hls(self,image):

return cv2.cvtColor(image,cv2.COLOR\_RGB2HLS)

def rgb\_to\_hsv(self,image):

return cv2.cvtColor(image,cv2.COLOR\_RGB2HSV)

# Helper function to perform perspective transform

def perspective\_transform(self,image):

src = np.float32([[250,400],[250,270],[550,270],[550,400]])

dst = np.float32([[250,400],[250,0],[550,0],[550,400]])

M = cv2.getPerspectiveTransform(src, dst)

Minv = cv2.getPerspectiveTransform(dst, src)

img\_size=(image.shape[1],image.shape[0])

warped = cv2.warpPerspective(image, M, img\_size, flags=cv2.INTER\_LINEAR)

return warped,Minv,M

def show\_image(self,image):

cv2.imshow("picture",image)

cv2.waitKey(1000)

cv2.destroyAllWindows()

def findCentroids(self,new\_dilate,rgb\_warped):

(\_,cnts,\_)= cv2.findContours(new\_dilate.copy(), cv2.RETR\_LIST, cv2.CHAIN\_APPROX\_SIMPLE)

for i in range(len(cnts)):

(x,y),radius = cv2.minEnclosingCircle(cnts[i])

center = (int(x),int(y))

radius = int(radius)

cv2.circle(rgb\_warped,center,radius,(0,255,0),2)

for i in range(len(cnts)):

M=cv2.moments(cnts[i])

cx = int(M['m10']/M['m00'])

cy = int(M['m01']/M['m00'])

l=[cx,cy]

self.centroids.append(l)

def run(self,img\_name):

# Read the image and perform necessary changes to the image

image = self.bgr\_to\_rgb(cv2.imread(img\_name))

image = cv2.resize(image,(800,400))

rgb\_image=image

gray\_image = rgb\_to\_gray(rgb\_image)

# Perform perspective transform on the image

rgb\_warped,Minv,M = self.perspective\_transform(rgb\_image)

gray\_warped,Minv,M = self.perspective\_transform(gray\_image)

rgb\_warped\_hsv = self.rgb\_to\_hsv(rgb\_warped)

rgb\_warped\_hsv\_blur = cv2.GaussianBlur(rgb\_warped\_hsv,(3,3),0)

rgb\_h = rgb\_warped\_hsv\_blur[:,:,0]

rgb\_s = rgb\_warped\_hsv\_blur[:,:,1]

ret,binary\_threshold\_s = cv2.threshold(rgb\_s,80,90,cv2.THRESH\_BINARY)

ret1,binary\_threshold\_h = cv2.threshold(rgb\_h,100,255,cv2.THRESH\_BINARY)

new = cv2.bitwise\_and(binary\_threshold\_s,binary\_threshold\_h)

kernel = np.ones((5,5),np.uint8)

new\_dilate = cv2.dilate(new,kernel,iterations=2)

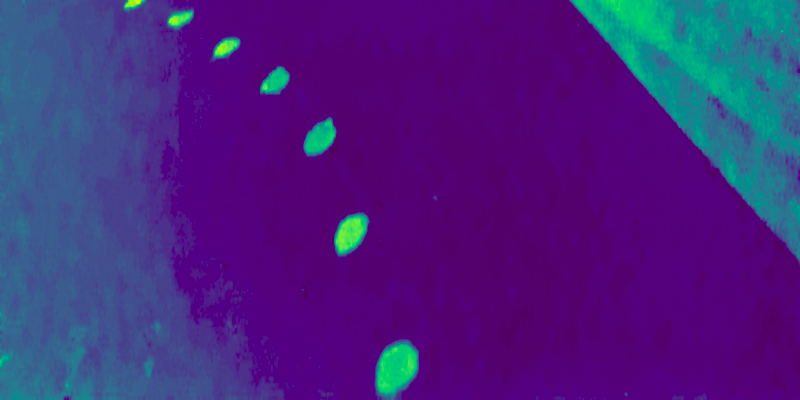
self.findCentroids(new\_dilate,rgb\_warped)

return self.centroids

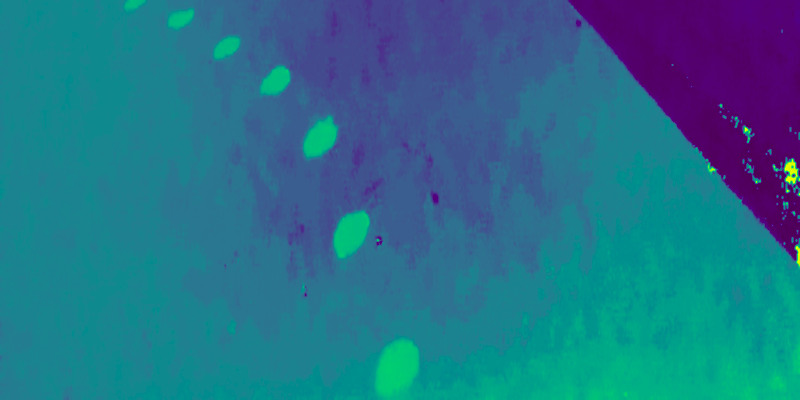
Results

We changed the color spaces of the images, and performed multiple transformations on the image.

We changed the color spaces of h and s color channel in HSV color space :

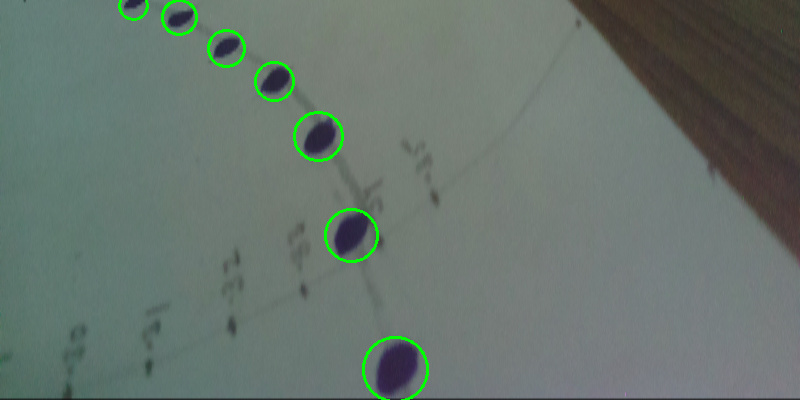


2.11 S channel of HSV color space



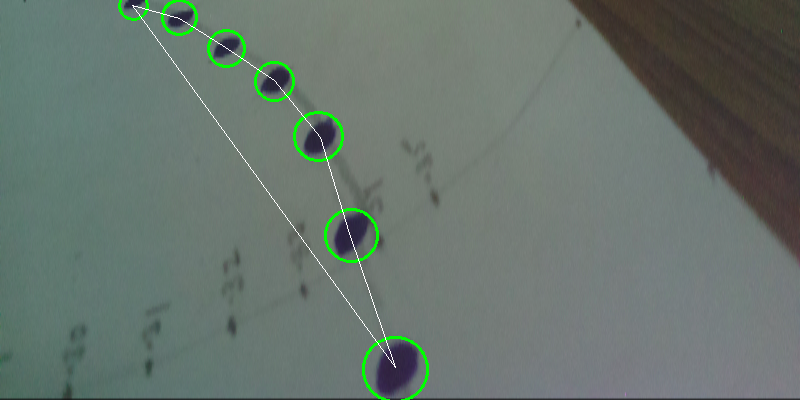
2.12 H channel of the HSV Colorspace

Finally the algorithm predicted the circles



2.13 Circles identified and marked on original image

Finally using the centres of the circles, we were able to fit a polynomial, which will become the reference trajectory



2.14 Polynomial fitted, to give the resultant desired path

# CHAPTER - 3

# MODEL PREDICTIVE CONTROL

# 

3.1 Model Predictive Control Introduction

Model Predictive Controls came into prominence in 1970s, but since the increase in the computation power in 2003, they have been intensively used in fields of control theory, process engineering and autonomous vehicles (very recently).

3.1.1 Idea behind Control Theory

Controls deals with the control of continuous operating dynamic systems, in engineered processes and machines.

Idea is to keep the machine in a controlled state based on some constraints, and making sure that the desired results are achieved and they do not overshoot.

3.1.2 MPC (Model Predictive Control)

It is an advanced method of process control, that is used to control a process while satisfying a set of constraints.

This is typically important when we have to modify a process, which directly or indirectly depends on some set of actuator controls

Model Predictive controls depends rely on the dynamic models of the process. The process can be a linear or a non-linear process, the nature of the constraints and the process adds to the task of difficulty of actually optimizing the process, to behave in a way that we desire it to.

The main advantage of MPC is that it allows the current time step to be optimized while keeping the future timesteps in account. MPC has the ability to predict the future and keep things in accordance to it.

This is the main difference between a PID controller and MPC, that PID actually does not keep into account the future states of the process.

3.2 Overview of MPC

Models used for an MPC, usually represent a complex dynamic system (in our case a moving car), the additional complexity that is provided by MPC is actually not always needed for simple systems, like heat room control etc., for those tasks, a simple PID controller will work efficiently. MPC models predict the change in the dependent variables of the modeled system that will be caused by changes in the independent variables. The independent processes that cannot be adjusted by the controller are used as disturbances. Dependent variables in these processes are other measurements that represent either control objectives or process constraints.

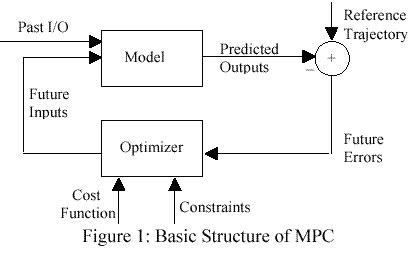
It is not necessary for the process to be a linear process in order for the MPC to work, we can model the problem using a non-linear process, either we can try to map the non-linear process onto a linear process and then proceed, or we can actually treat it as a non linear optimization problem. The nonlinear model may be linearized to derive a Kalman filter or specify a model for linear MPC.

Figure 3.1

Source: https://www.plantautomation.com/doc/asimplifiedandintegratedapproachtomodel001

3.3 Theory behind MPC

MPC is based on an iterative, finite horizon of a particular dynamic model. At time **t,** the current state of the object(vehicle in our case) is sampled, and a cost minimizing control strategy is computed (this is achieved with the help of linear on non-linear optimization which is usually a minimization problem) for a relatively short horizon in the future **[t,t+T]**.

Now what happens is either an online or on-the-fly calculations are used to explore the state trajectories that emanate from the current state, and find (via solutions like the Euler-Lagrange equations, KKT, or any other solver for the optimization problems) a cost minimizing control strategy till **[t,t+T]**. Only the first state is implemented, then again, the states are sampled and the calculations are repeated starting from the new current state, yielding a new current state and a new predicted state path. The prediction horizon keeps on moving forward and for this reason it is also called the **receding horizon control**.Although this approach is not optimal, but because of the advances in the computations and the feasibility of the implementations, this yields a very decent set of results.

3.3.1 Principles of MPC

MPC is a multivariable control algorithm that uses:

* An internal dynamic model of the process (kinematic model)
* A history of past control moves
* An optimal cost function J over receding prediction horizon

To control the optimal control moves.

An example of non-linear cost function for optimization is given by:

J = wxi(ri - xi)2 + wuiΔui2 (3.1)

xi : is the vector of the current states

ri : is the reference state (if any)

ui : ith actuator variable

wxi : weighted coefficients reflection the importance of xi

wui : weighted coefficients penalizing relative big changes in ui

3.4 Vehicle Model

We are using the **Kinematic bicycle model**, to model the dynamics of our car

Kinematic models, are the simplification of the dynamic models, in this we actually ignore the tyre forces, friction, gravity and mass of the car. The main idea behind doing this is, that we want the model to be as general as possible. This enables us to write a very generalized set of controls so that we do not actually have to rewrite every aspect of the control and vehicle model redundantly. The main advantage is that, this allows a very agnostic software for the self driving car platforms.

The main idea is to store the state of the vehicle in a vector, that is called the state vector or xi

Xi is a 4 tuple of the form : [x\_position, y\_position, absolute velocity, orientation of the car]

* x\_position gives the x coordinates of the car,from a reference
* y\_position gives the y coordinates of the car, from a reference
* absolute velocity - is the speed of the car
* orientation of the car : it is the anti clockwise angle of the car, from the reference

Ui is a 2 tuple, which stores the control variable of the car, namely the acceleration and the steering angle (in our model)

Ui = [delta,a]

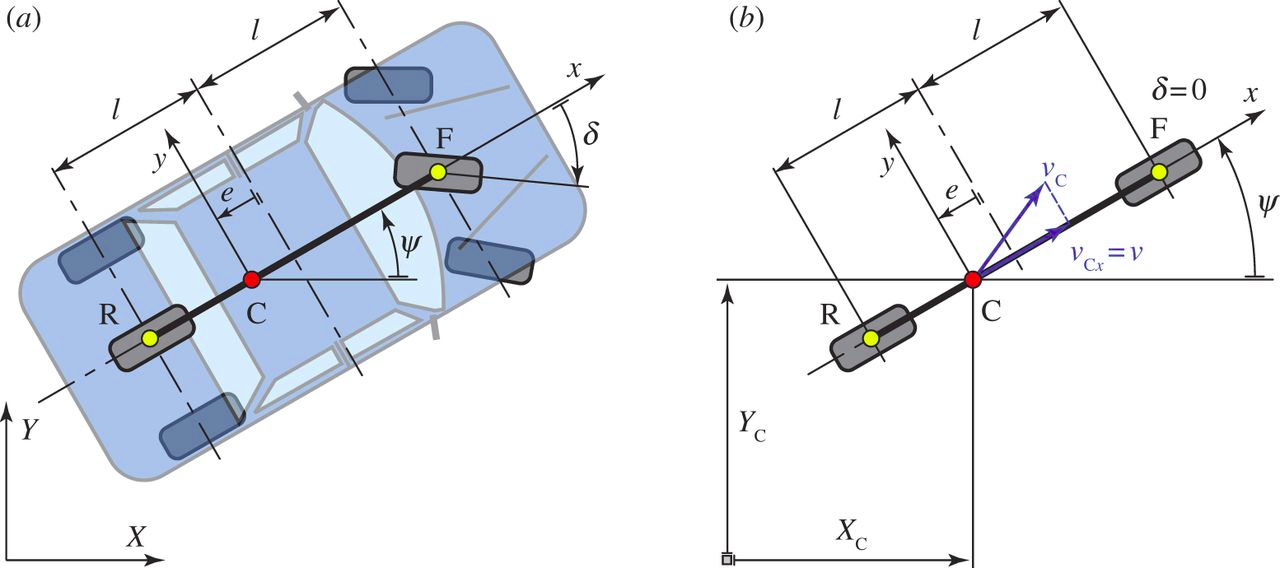
* delta : steering angle of the car
* a : actuation/ acceleration of the car

Figure 3.2

Source: <http://rsta.royalsocietypublishing.org/content/371/1993/20120427>

The speed of the car is modelled as

**v = v + a \* dt**

The rest of the equations based on the vehicle model of the car are :

xt+1 = xt + vt \* cos(Ψt) \* dt

yt+1 = yt + vt \* sin(Ψt) \* dt

Ψt+1  = Ψt + vt / Lf \* δt \* dt

Vt+1 = vt + a \* dt

3.5 Constraints

The controls that we are taking into consideration, are actually constrained between some particular values, these constraints actually form the basis of the optimization problem that we have to solve. We have to minimize the cost of the movement of the car, subject to these constraints.

The constraints that we are considering for this particular model are :

* **delta** (steering angle) : the steering angle of the car should be between a particular value, that is the tyres actually have a restricted movement, and not a free 360o movement. For the purpose of this project, we consider that the tyres are constrained to turn between angles [-300,-300].

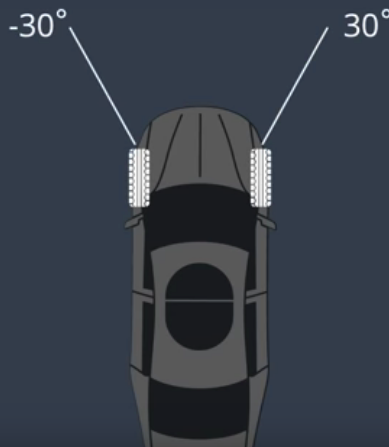


Figure 3.3

Source: [www.udacity.com](http://www.udacity.com) - Self Driving Cars Nanodegree

* **actuator** (motor acceleration) : the motor in the car, that is attached has a maximum and a minimum limit. As we do not have the facility of brakes, we have to constraints that the motor acceleration is between 0 and 1, that is, the **a >= 0** and **a <= 1.**

Code

// The upper and lower limits of delta are set to -30 and 30

// degrees (values in radians).

// NOTE: Feel free to change this to something else.

**for (int i = delta\_start; i < a\_start; i++) {**

**vars\_lowerbound[i] = -0.436332;**

**vars\_upperbound[i] = 0.436332;**

**}**

***> Here the values -0.436332 is actually -250 in radians***

// Acceleration/deceleration upper and lower limits.

// NOTE: Feel free to change this to something else.

**for (int i = a\_start; i < n\_vars; i++) {**

**vars\_lowerbound[i] = -1.0;**

**vars\_upperbound[i] = 1.0;**

**}**

3.6 Reference state

The task that we have in hand is that we have to make a car move on a desired track, based on some reference path, being controlled by some actuators which have some constraints.

Now, the references that are taken into consideration are actually the **orientation error** of the car and the **cross track error**

* **Orientation error**

**This is the difference in the orientation of the car from the actual direction of the desired path**.

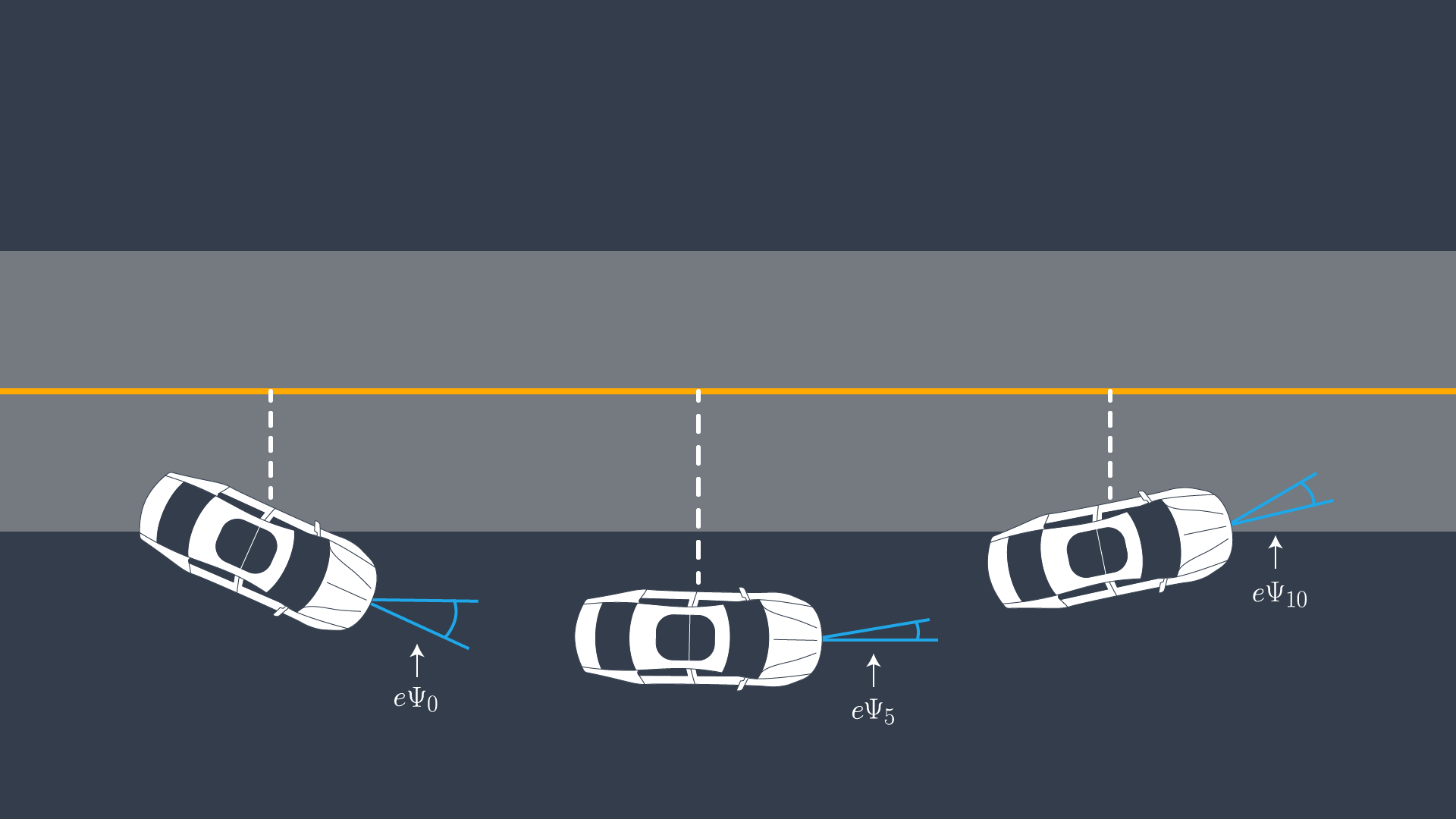


Figure 3.4

Source: Self-Driving Car Nanodegree

eΨt - is the desired orientation subtracted from the from the current orientation

eΨt  = Ψt - Ψdest

The update rule of the orientation error, goes as :

eΨt+1 = eΨt  + vt/Lf \* (𝛿t) \* dt

Replacing the values in this equation we get

eΨt+1 = Ψt - Ψdest  + vt/Lf \* (𝛿t) \* dt

* **Cross Track Error** :

It is the distance between the current position of the car, to the desired path. We need to maintain the motion of the car, in a particular path, which is achieved with the help of cross track error.

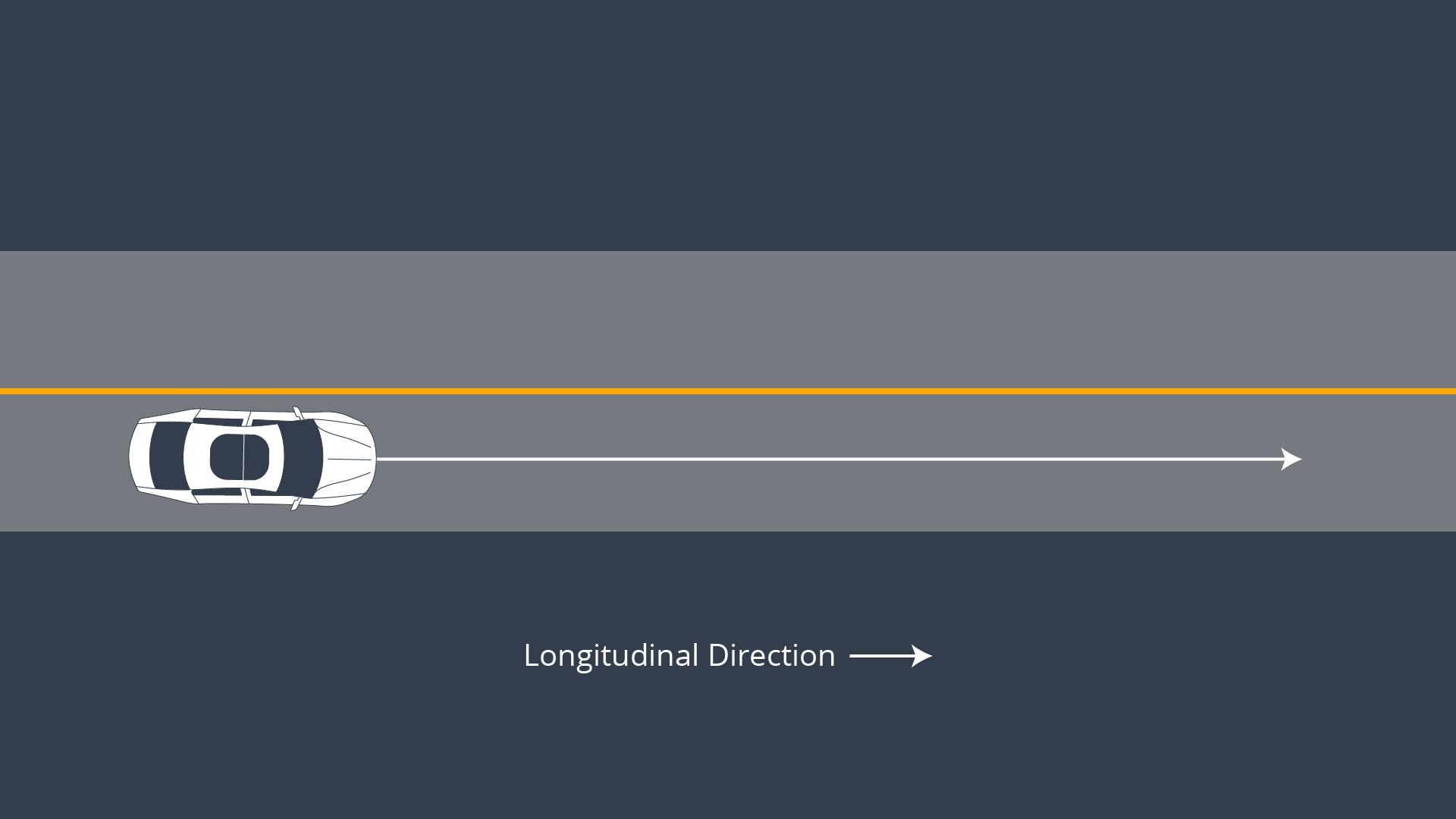


Figure 3.5

Source: Self-Driving Car Nanodegree

At t+1 time, the cross track error (cte) is given by:

ctet+1 = ctet + vt \* sin(eΨt) \* dt

cte at the tth time step can be determined, let the reference path is given by a polynomial ft :

ctet  = yt - f(xt)

replacing the values in the above equation we get

ctet+1 = (yt - f(xt)) + vt \* sin(eΨt) \* dt

3.7 Optimization

The crux of the problem that we are left with at each time step is that we have to

minimize the cost : (ctet)2  + (eΨt)2

constraints :

0 <= a <= 1

-0.436332 <= delta <= 0.436332

Now, this problem actually looks like that it has a quadratic nature, but for the simplicity of the implementation and a generalized algorithm, we consider the non-linear convex optimization method called the interior point optimization or the boundary method.

For the implementation we used two libraries :

1. IPOPT - optimization library for C++
2. CVXPY - Convex optimization library for Python

This optimizations takes place at each and every timestep. And we get the desired actuations for the cars

3.8 Simulation Results

The simulations of the model, were run on the Udacity Self Driving car simulator, which is an open source simulator to test some of the codes, which are written specific to the self driving cars.

C++ code

#include "MPC.h"

#include <cppad/cppad.hpp>

#include <cppad/ipopt/solve.hpp>

#include "Eigen-3.3/Eigen/Core"

using CppAD::AD;

// TODO: Set the timestep length and duration

size\_t N = 10;

double dt = 0.05;

double latency = 0.1;

double dt\_with\_latency = dt + latency;

// This value assumes the model presented in the classroom is used.

//

// It was obtained by measuring the radius formed by running the vehicle in the

// simulator around in a circle with a constant steering angle and velocity on a

// flat terrain.

//

// Lf was tuned until the the radius formed by the simulating the model

// presented in the classroom matched the previous radius.

//

// This is the length from front to CoG that has a similar radius.

const double Lf = 2.67;

// the reference cross track error and the reference orientation is zero

// the reference velocity is set to 80mph

double ref\_velocity = 80;

// the solver takes all the state and actuator variables as a single

// vector, so we will keep a track of the starting and the ending

// variables

size\_t x\_start = 0;

size\_t y\_start = x\_start + N;

size\_t psi\_start = y\_start + N;

size\_t v\_start = psi\_start + N;

size\_t cte\_start = v\_start + N;

size\_t epsi\_start = cte\_start + N;

size\_t delta\_start = epsi\_start + N;

size\_t a\_start = delta\_start + N - 1;

class FG\_eval {

public:

// Fitted polynomial coefficients

Eigen::VectorXd coeffs;

FG\_eval(Eigen::VectorXd coeffs) { this->coeffs = coeffs; }

typedef CPPAD\_TESTVECTOR(AD<double>) ADvector;

void operator()(ADvector& fg, const ADvector& vars) {

// TODO: implement MPC

// `fg` a vector of the cost constraints, `vars` is a vector of variable values (state & actuators)

// NOTE: You'll probably go back and forth between this function and

// the Solver function below.

// the first value in the vector stores the cost

fg[0] = 0;

// cost based on the reference state

for(int t = 0 ; t < N ; t++){

fg[0] += 1500\*CppAD::pow(vars[cte\_start + t],2);

fg[0] += 1500\*CppAD::pow(vars[epsi\_start + t],2);

fg[0] += CppAD::pow(vars[v\_start + t] - ref\_velocity,2);

// Minimize the use of actuators --> We want as smooth a ride as possible

if(t < N-1){

fg[0] += 20\*CppAD::pow(vars[delta\_start + t], 2);

fg[0] += 40\*CppAD::pow(vars[a\_start + t], 2);

}

// for smooth response to the actuators, we will reduce the difference between two actuator commands

if(t < N-2){

fg[0] += 200\*CppAD::pow(vars[delta\_start + t + 1] - vars[delta\_start + t], 2);

fg[0] += 100\*CppAD::pow(vars[a\_start + t + 1] - vars[a\_start + t], 2);

}

}

// The coefficients that are mutliplied have been attained from some random trying and testing

// also some help from the forums was taken

// We will now model the constraints

// initial constrains, we will add 1 because the first entry is reserved for the error

fg[1 + x\_start] = vars[x\_start];

fg[1 + y\_start] = vars[y\_start];

fg[1 + psi\_start] = vars[psi\_start];

fg[1 + v\_start] = vars[v\_start];

fg[1 + cte\_start] = vars[cte\_start];

fg[1 + epsi\_start] = vars[epsi\_start];

// rest of the constraints

for (int i = 0; i < N - 1; i++) {

AD<double> x1 = vars[x\_start + i + 1];

AD<double> y1 = vars[y\_start + i + 1];

AD<double> psi1 = vars[psi\_start + i + 1];

AD<double> v1 = vars[v\_start + i + 1];

AD<double> cte1 = vars[cte\_start + i + 1];

AD<double> epsi1 = vars[epsi\_start + i + 1];

AD<double> x0 = vars[x\_start + i];

AD<double> y0 = vars[y\_start + i];

AD<double> psi0 = vars[psi\_start + i];

AD<double> v0 = vars[v\_start + i];

AD<double> cte0 = vars[cte\_start + i];

AD<double> epsi0 = vars[epsi\_start + i];

AD<double> delta0 = vars[delta\_start + i];

AD<double> a0 = vars[a\_start + i];

// we will be fitting a three degree polynomial

AD<double> f0 = coeffs[0] + coeffs[1] \* x0 + coeffs[2] \* x0 \* x0 + coeffs[3] \* x0 \* x0 \* x0;

AD<double> psides0 = CppAD::atan(coeffs[1] + 2 \* coeffs[2] \* x0 + 3 \* coeffs[3] \* x0 \* x0);

// The idea here is to constraint this value to be 0.

//

// NOTE: The use of `AD<double>` and use of `CppAD`!

// This is also CppAD can compute derivatives and pass

// these to the solver.

// constraints are inserted into the vector, in the correct representation as in the model

// we are taking into consideration the dt values with latency, so we use dt\_with\_latency

fg[2 + x\_start + i] = x1 - (x0 + v0 \* CppAD::cos(psi0) \* dt\_with\_latency);

fg[2 + y\_start + i] = y1 - (y0 + v0 \* CppAD::sin(psi0) \* dt\_with\_latency);

fg[2 + psi\_start + i] = psi1 - (psi0 + v0/Lf \* delta0 \* dt\_with\_latency);

fg[2 + v\_start + i] = v1 - (v0 + a0 \* dt\_with\_latency);

fg[2 + cte\_start + i] = cte1 - ((f0 - y0) + v0 \* CppAD::sin(epsi0) \* dt\_with\_latency);

fg[2 + epsi\_start + i] = epsi1 - (psi0 - psides0 + v0 / Lf \* delta0 \* dt\_with\_latency);

}

}

};

//

// MPC class definition implementation.

//

MPC::MPC() {}

MPC::~MPC() {}

vector<double> MPC::Solve(Eigen::VectorXd x0, Eigen::VectorXd coeffs) {

bool ok = true;

size\_t i;

typedef CPPAD\_TESTVECTOR(double) Dvector;

double x = x0[0];

double y = x0[1];

double psi = x0[2];

double v = x0[3];

double cte = x0[4];

double epsi = x0[5];

// TODO: Set the number of model variables (includes both states and inputs).

// For example: If the state is a 4 element vector, the actuators is a 2

// element vector and there are 10 timesteps. The number of variables is:

//

// appropriate number of constraints according to the model

// 4 \* 10 + 2 \* 9

size\_t n\_vars = N\*6 + (N-1)\*2;

// TODO: Set the number of constraints

size\_t n\_constraints = N\*6;

// Initial value of the independent variables.

// SHOULD BE 0 besides initial state.

Dvector vars(n\_vars);

for (int i = 0; i < n\_vars; i++) {

vars[i] = 0.0;

}

Dvector vars\_lowerbound(n\_vars);

Dvector vars\_upperbound(n\_vars);

// TODO: Set lower and upper limits for variables.

// Set all non-actuators upper and lowerlimits

// to the max negative and positive values.

// bounds as mentioned in the theory and the previous exercise

for (int i = 0; i < delta\_start; i++) {

vars\_lowerbound[i] = -1.0e19;

vars\_upperbound[i] = 1.0e19;

}

// The upper and lower limits of delta are set to -25 and 25

// degrees (values in radians).

// NOTE: Feel free to change this to something else.

for (int i = delta\_start; i < a\_start; i++) {

vars\_lowerbound[i] = -0.436332;

vars\_upperbound[i] = 0.436332;

}

// Acceleration/decceleration upper and lower limits.

// NOTE: Feel free to change this to something else.

for (int i = a\_start; i < n\_vars; i++) {

vars\_lowerbound[i] = -1.0;

vars\_upperbound[i] = 1.0;

}

// Lower and upper limits for the constraints

// Should be 0 besides initial state.

Dvector constraints\_lowerbound(n\_constraints);

Dvector constraints\_upperbound(n\_constraints);

for (int i = 0; i < n\_constraints; i++) {

constraints\_lowerbound[i] = 0;

constraints\_upperbound[i] = 0;

}

constraints\_lowerbound[x\_start] = x;

constraints\_lowerbound[y\_start] = y;

constraints\_lowerbound[psi\_start] = psi;

constraints\_lowerbound[v\_start] = v;

constraints\_lowerbound[cte\_start] = cte;

constraints\_lowerbound[epsi\_start] = epsi;

constraints\_upperbound[x\_start] = x;

constraints\_upperbound[y\_start] = y;

constraints\_upperbound[psi\_start] = psi;

constraints\_upperbound[v\_start] = v;

constraints\_upperbound[cte\_start] = cte;

constraints\_upperbound[epsi\_start] = epsi;

// object that computes objective and constraints

FG\_eval fg\_eval(coeffs);

//

// NOTE: You don't have to worry about these options

//

// options for IPOPT solver

std::string options;

// Uncomment this if you'd like more print information

options += "Integer print\_level 0\n";

// NOTE: Setting sparse to true allows the solver to take advantage

// of sparse routines, this makes the computation MUCH FASTER. If you

// can uncomment 1 of these and see if it makes a difference or not but

// if you uncomment both the computation time should go up in orders of

// magnitude.

options += "Sparse true forward\n";

options += "Sparse true reverse\n";

// NOTE: Currently the solver has a maximum time limit of 0.5 seconds.

// Change this as you see fit.

options += "Numeric max\_cpu\_time 50.5\n"; // The value changed to suit my specifications

// place to return solution

CppAD::ipopt::solve\_result<Dvector> solution;

// solve the problem

CppAD::ipopt::solve<Dvector, FG\_eval>(

options, vars, vars\_lowerbound, vars\_upperbound, constraints\_lowerbound,

constraints\_upperbound, fg\_eval, solution);

// Check some of the solution values

ok &= solution.status == CppAD::ipopt::solve\_result<Dvector>::success;

// Cost

auto cost = solution.obj\_value;

std::cout << "Cost " << cost << std::endl;

// TODO: Return the first actuator values. The variables can be accessed with

// `solution.x[i]`.

//

// {...} is shorthand for creating a vector, so auto x1 = {1.0,2.0}

// creates a 2 element double vector.

vector<double> result = {solution.x[delta\_start], solution.x[a\_start]};

for (int i = 0; i< 2 \* N; i++){

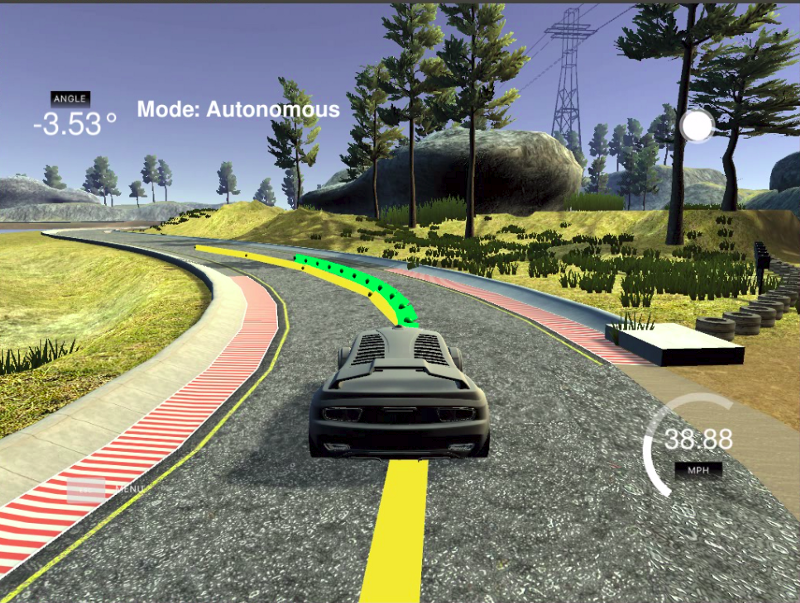
result.push\_back(solution.x[x\_start+i]);

}

return result;

}

Results



The car, did not leave the track in the entirety of the circular track.

For the purpose of running this code in hardwares, the code had to be ported into Python, as the RaspberryPi, had some issues with the library implementations of IPOPT and CPPAD which were necessary for the model.

For the same we used CVXPY library

import cvxpy

import numpy as np

from cvxpy import \*

#import matplotlib.pyplot as plt

from math import \*

import time

class MPC:

dt = 0

max\_s =0

min\_s =0

lr =0

ref\_vel= 3

def \_\_init\_\_(self,max\_s,min\_s,dt,lr,ref\_vel):

self.dt = dt

self.lr =lr

self.max\_s = max\_s

self.min\_s= min\_s

self.ref\_vel= ref\_vel

def LinealizeCarModel(self,xb, u,coeffs):

u"""

TODO conplete model

"""

print(coeffs)

x = xb[0]

y = xb[1]

v = xb[2]

theta = xb[3]

cte = xb[4]

epsi = xb[5]

a = u[0]

beta = u[1]

f0 = (coeffs[0]\*x\*x + coeffs[1] \* x + coeffs[2])\*20/75591

psides0 = atan((2\*coeffs[0]\*x + coeffs[1])\*20/75591)

t1 = self.dt \* v \* sin(theta + beta)

t2 = -self.dt \* v \* cos(theta + beta)

A = np.eye(xb.shape[0])

A[0, 2] = self.dt \* cos(theta + beta)

A[1, 2] = self.dt \* sin(theta + beta)

A[3, 2] = self.dt \* sin(beta) / self.lr

A[0, 3] = t1

A[1, 3] = t2

A[4, 4] = 0

A[4, 2] = sin(epsi) \* self.dt

A[5, 5] = 0

B = np.zeros((xb.shape[0], u.shape[0]))

B[2, 0] = self.dt

B[0, 1] = t1

B[1, 1] = t2

B[3, 1] = self.dt \* v \* cos(beta) / self.lr

B[5, 1] = v/self.lr \*self.dt

tm = np.zeros((6, 1))

tm[0, 0] = v \* cos(theta + beta) \* self.dt

tm[1, 0] = v \* sin(theta + beta) \* self.dt

tm[2, 0] = a \* self.dt

tm[3, 0] = v / self.lr \* sin(beta) \* self.dt

tm[4, 0] = (f0 - y) + v \* sin(epsi) \* self.dt

tm[5, 0] = (theta - psides0) + v/self.lr \* beta\*self.dt

C = xb + tm

C = C - A \* xb - B \* u

return A, B, C

def NonlinearModel(self,x,u,coeffs):

# print(x[0].value)

# print(coeffs)

# x\_0 = x

# f0 = coeffs[0]\*x[0]\*x[0] + coeffs[1] \* x[0] + coeffs[2]

# print(f0)

# psides0 = atan(float(2\*coeffs[0]\*x[0] + coeffs[1]))

# print(x[0].value)

# print(x[1])

# print(x[2])

# print(x[3])

x[0] = x[0] + x[2] \* cos(x[3] + u[1]) \* self.dt

x[1] = x[1] + x[2] \* sin(x[3] + u[1]) \* self.dt

x[2] = x[2] + u[0] \* self.dt

x[3] = x[3] + x[2] / self.lr \* sin(u[1]) \* self.dt

# x[4] = ((f0 - x\_0[1]) + x\_0[2] \* sin(x\_0[5]) \* self.dt)

# x[5] = (x\_0[3]-psides0) + x\_0[2]/self.lr \* u[1]\*self.dt

return x

def CalcInput(self,A,B,C,x, u,T,coeffs):

states=[]

x\_0=x[:]

# x\_1 = x\_0

# u\_1 = u

x = Variable(x.shape[0],T+1)

u = Variable(u.shape[0],T)

for t in range(T):

# A, B, C = mpc.LinealizeCarModel(x\_1,u\_1,coeffs)

constr = [x[:, t + 1] == A \* x[:, t] + B \* u[:, t] + C]

print(x[:,t+1].value)

# constr = [x[:,t+1] == self.NonlinearModel(x[:,t],u[:,t],coeffs)]

constr += [x[2, t + 1] <= self.max\_s]

constr += [x[2, t + 1] >= self.min\_s]

constr += [u[0 ,t ] <=1]

constr += [u[0 ,t ] >=0]

constr += [u[1 ,t ] >=-0.436332]

constr += [u[1 ,t ] <= 0.436332]

cost = 150\*x[4,t] \*\*2

cost += 150\*x[5,t]\*\*2

cost += (x[2,t]-self.ref\_vel)\*\*2

states.append(Problem(Minimize(cost), constr))

prob = sum(states)

prob.constraints += [x[:, 0] == x\_0,x[2, T] == 0.0]

result = prob.solve()

print(prob.value)

print(u.value)

return u,x,prob.value

# CHAPTER - 4

# HARDWARE

# 

4.1 Raspberry Pi

A Raspberry Pi is a general-purpose computer, usually with a Linux operating system, and the ability to run multiple programs. It is more complicated to use than an Arduino. Raspberry Pi is best used when we need a full-fledged computer: driving a more complicated robot, performing multiple tasks and doing intense calculations. The Raspberry Pi 3 Model B is the third generation Raspberry Pi. This powerful credit-card sized single board computer can be used for many applications and supersedes the original Raspberry Pi Model B+ and Raspberry Pi 2 Model B. Whilst maintaining the popular board format the Raspberry Pi 3 Model B brings a more powerful processor, 10x faster than the first generation Raspberry Pi. Additionally it adds wireless LAN & Bluetooth connectivity making it the ideal solution for powerful connected designs.

Raspberry Pi 3 - Model B Technical Specifications:

* Broadcom BCM2387 chipset
* 1.2GHz Quad-Core ARM Cortex-A53
* 802.11 bgn Wireless LAN and Bluetooth 4.1 (Bluetooth Classic and LE)
* 1.2GHz Quad-Core ARM Cortex-A53
* 64 Bit CPU
* 1GB RAM
* 4 x USB ports
* 4 pole Stereo output and Composite video port
* Full size HDMI
* CSI camera port for connecting the Raspberry Pi camera
* Micro USB power source

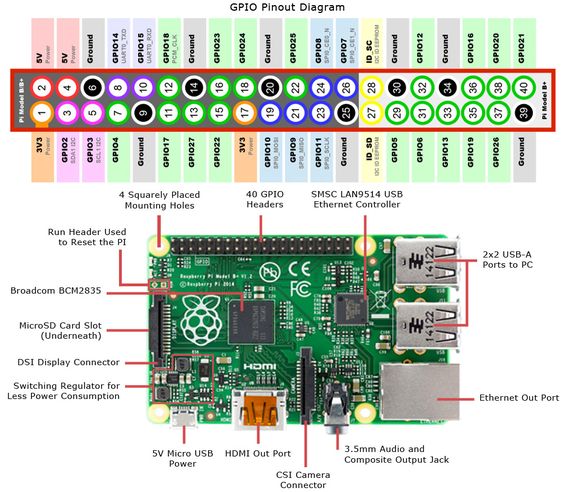


Figure 4.1

Source: <https://www.pinterest.fr/pin/415457134368613757/>

4.2 Brushed DC motor

A brushed DC motor is an internally [commutated](https://en.wikipedia.org/wiki/Commutator_(electric)) [electric motor](https://en.wikipedia.org/wiki/Electric_motor) designed to be run from a [direct current](https://en.wikipedia.org/wiki/Direct_current) power source. Brushed motors were the first commercially important application of electric power to driving mechanical energy, and DC distribution systems were used for more than 100 years to operate motors in commercial and industrial buildings. Brushed DC motors can be varied in speed by changing the operating voltage or the strength of the magnetic field. Depending on the connections of the field to the power supply, the speed and torque characteristics of a brushed motor can be altered to provide steady speed or speed inversely proportional to the mechanical load. Brushed motors continue to be used for electrical propulsion, cranes, paper machines and steel rolling mills. Since the brushes wear down and require replacement, [brushless DC motors](https://en.wikipedia.org/wiki/Brushless_DC_electric_motor) using [power electronic devices](https://en.wikipedia.org/wiki/Power_electronics) have displaced brushed motors from many applications.

When a current passes through the coil wound around a soft iron core, the side of the positive pole is acted upon by an upwards force, while the other side is acted upon by a downward force. According to [Fleming's left hand rule](https://en.wikipedia.org/wiki/Fleming%27s_left_hand_rule), the forces cause a turning effect on the coil, making it rotate. To make the motor rotate in a constant direction, "direct current" commutators make the current reverse in direction every half a cycle (in a two-pole motor) thus causing the motor to continue to rotate in the same direction.

A problem with the motor shown above is that when the plane of the coil is parallel to the magnetic field—i.e. when the rotor poles are 90 degrees from the stator poles—the torque is zero. In the pictures above, this occurs when the core of the coil is horizontal—the position it is just about to reach in the last picture on the right. The motor would not be able to start in this position. However, once it was started, it would continue to rotate through this position by momentum.

4.2.1 Pulse Width Modulation (PWM)

Motor speed control of DC motor is nothing new. A simplest method to control the rotation speed of a DC motor is to control its driving voltage. The higher the voltage is the higher speed the motor tries to reach. In many applications a simple voltage regulation would cause lots of power loss on control circuit, so a pulse width modulation method (PWM) is used in many DC motor controlling applications. In the basic Pulse Width Modulation (PWM) method, the operating power to the motors is turned on and off to modulate the current to the motor. The ratio of "on" time to "off" time is what determines the speed of the motor. When doing PWM controlling, keep in mind that a motor is a low pass device. The reason is that a motor is mainly a large inductor. It is not capable of passing high frequency energy, and hence will not perform

well using high frequencies. Reasonably low frequencies are required, and then PWM techniques will work. Lower frequencies are generally better than higher frequencies, but PWM stops being effective at too low a frequency. The idea that a lower frequency PWM works better simply reflects that the "on" cycle needs to be pretty wide before the motor will draw any current because of motor inductance). A higher PWM frequency will work fine if you hang a large capacitor across the motor or short the motor out on the "off" cycle (e.g. power/brake PWM). The reason for this is that short pulses will not allow much current to flow before being cut off. Then the current that did flow is dissipated as an inductive kick - probably as heat through the fly-back diodes. The capacitor integrates the pulse and provides a longer, but lower, current flow through the motor after the driver is cut off. There is not inductive kick either, since the current flow isn't being cut off. Knowing the low pass roll-off frequency of the motor helps to determine an optimum frequency for operating PWM.

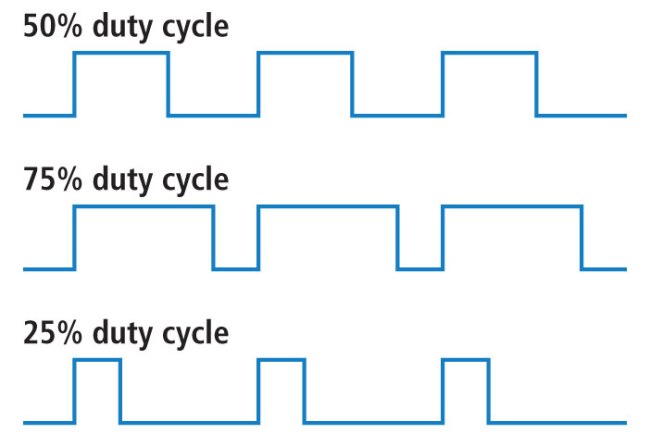


Figure 4.2

Source: <https://learn.sparkfun.com/tutorials/pulse-width-modulation>

4.3 H - Bridge circuit

An H bridge is an electronic circuit that enables a voltage to be applied across a load in either direction. These circuits are often used in robotics and other applications to allow DC motors to run forwards or backwards. Most DC-to-AC converters (power inverters), most AC/AC converters, the DC-to-DC push–pull converter, most motor controllers, and many other kinds of power electronics use H bridges. In particular, a bipolar stepper motor is almost invariably driven by a motor controller containing Two H Bridges. H bridges are available as integrated circuits, or can be built from discrete components. The term H bridge is derived from the typical graphical representation of such a circuit. An H bridge is built with four switches (solid-state or mechanical). When the switches S1 and S4 (according to the first figure) are closed (and S2 and S3 are open) a positive voltage will be applied across the motor. By opening S1 and S4 switches and closing S2 and S3 switches, this voltage is reversed, allowing reverse operation of the motor.

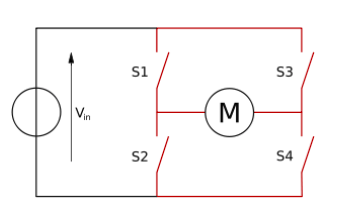


Figure 4.3

Source: Self-made

# REFERENCES

# 

[1]. Computer Graphics and Multimedia Donald Hearn & Pauline Baker, Page number 572-580.

[2]. <https://docs.opencv.org/2.4.13/>

[3]. <https://learn.sparkfun.com/tutorials/pulse-width-modulation>

[4]. <https://www.tutorialspoint.com/dip/perspective_transformation.htm>

[5]. Computer Graphics, Second Edition, by Zhigang Xiang and Roy Plastock, Schaum Series. Pages: 229-235

[6]. <https://en.wikipedia.org/wiki/Model_predictive_control>

[7]. Udacity Self Driving Cars Nanodegree, course material

[8]. <http://www.mpc.berkeley.edu/mpc-course-material>

[9]. CVXPY - Convex Optimization library for Python

[10]. IPOPT - C++ library for optimization

[11]. Udacity Self Driving Cars Term 2 simulator

# 