Conceptualization on Autonomous Parking for Electric Vehicle Battery Charging and its Future Works

Korea University Alumni

Bruce KwangKyun Kim | IlGyu Cho | Limaries HanDong Lim | HoYong Park | KunHee Han

---

Objective

Our team believes that the era of autonomous vehicles is in its early stage, and so are the related markets. As more vehicles drive autonomously, we expect creation and growth of the related markets including autonomous battery charging service. This project aims to 1) study and analyze the latest autonomous parking algorithms for application, 2) implement and simulate autonomous parking, and 3) conceptualize business idea of autonomous charging service including a mobile application and a battery charger robot manipulator. The service is to enable self-driving cars charge their batteries instead of letting cars merely idle at parking lots. When drivers or car owners search and designate charging stations via web or mobile, cars would autonomously drive to the chosen station and charge their batteries. This allows people to focus on their affairs or work in the office without concerning of charging their low-battery cars in the charging station after work and wasting their precious time on the road. Mobile application to match vehicle owners with charging stations and robot arm concepts to plug the battery charger will be discussed in the future work. The projected started in July 2019 and ended in January 2021. The entire progress of the work is uploaded on the following websites: https://github.com/limaries30/AutoParking (https://github.com/brucekimrok/AutoParking).

Introduction

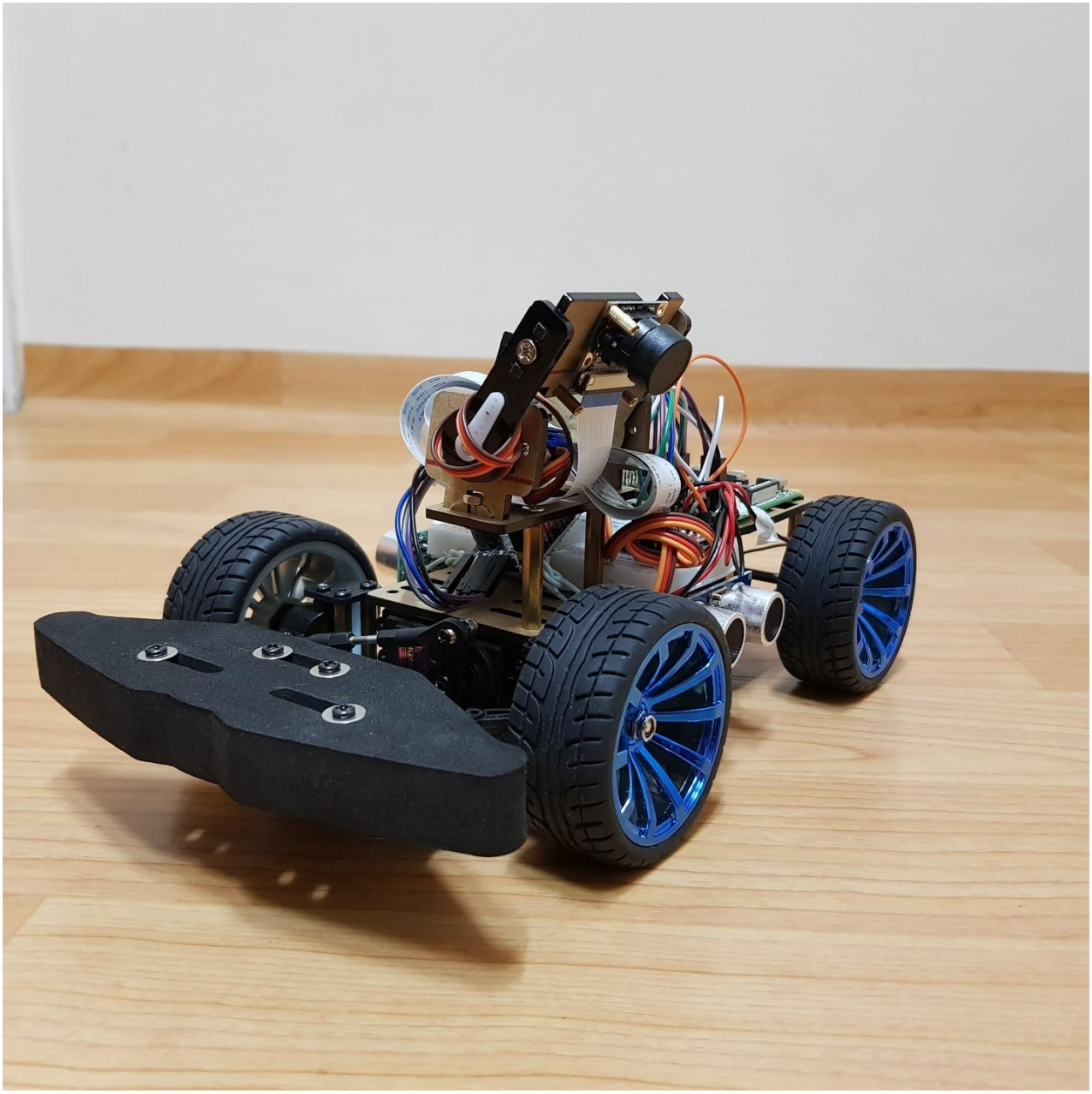


Figure 1. Autonomous Parking Car

This project is about conceptualizing an autonomous parking vehicle, which is programed to perceive its environment and create birds-view images, identify types of parking slot, process spatial information of the detected parking slot, and park safely. As the level of the project is mainly conceptualization, our team has utilized the latest algorithms, which suit the purpose of the project the most.

The autonomous parking car is mainly designed to run on a Raspberry Pi ver.4B (RPi), which integrates image processing and steering software with GPIO I/O regulation. The RPi serves as the main controller for the entire system. The hardware constraints of the vehicle are set under the notion to resemble typical mass-produced cars by automotive OEMs as much as possible. While adding new types of sensor may be helpful in making the technical challenge easier, additional parts means the increase in mass production costs, thereby reducing the odds of

realization in the existing industry. Therefore, our vehicle tried to adopt sensors used in nowadays electric autonomous vehicles, such as that of Tesla as it is portrayed in the below image.

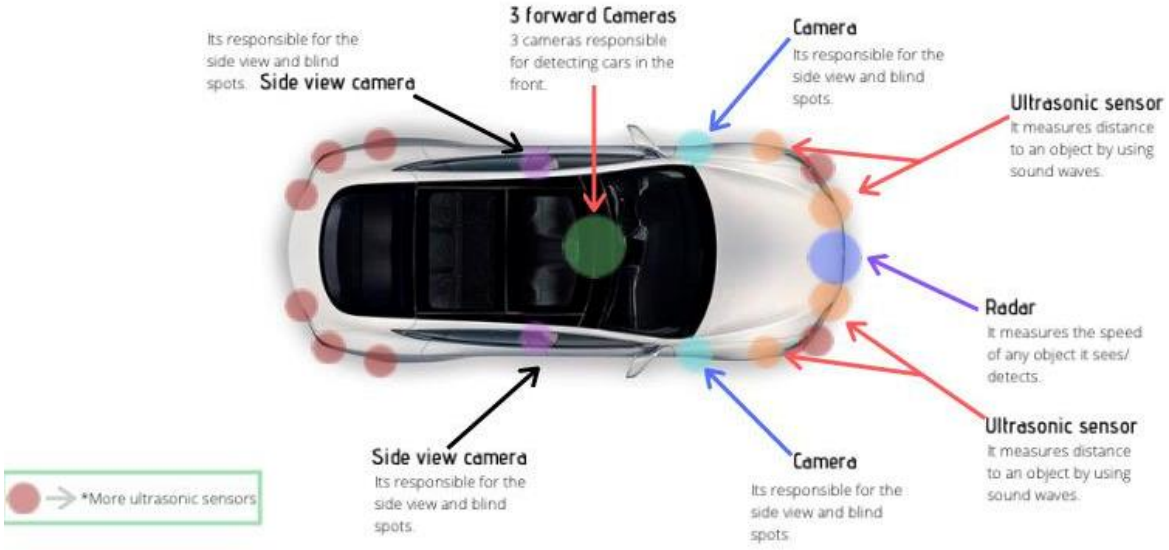


Figure 2. Sensors on Tesla Model 3

Components of the vehicle include one vehicle chassis with a suspension structure, two RPi fisheye camera lens modules, one Arducam Multi Camera Adapter Doubleplexer Stereo module, one PCA9685 PWM/Servo driver, one MG996R servo motor, one TB6612 DC/Stepper motor driver, one 12V DC motor, and three time-of-flight (TOF) ultrasonic sensors. Fisheye cameras are used as image sensors for Around View Monitoring (AVM) system. Each camera has 200° angle of view and it mainly captures the surrounding environment as input data for AVM. After cameras take shots of the surrounding area, AVM software processes and converts the images into a birds-view image.

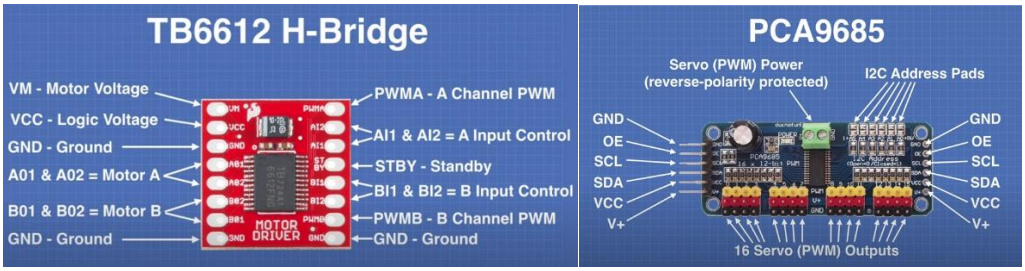


Figure 3. DC Motor Driver & Servo Motor Driver

The birds-view image of nearby parking lot is then provided as input data for parking slot detection algorithm. The algorithm we use is a context-based parking slot detection method. Parking context recognizer identifies the parking environment consisting of type, angle, and availability of a parking slot. Then the parking slot detector locates the exact dimension of the parking slot.

After the vehicle completes the identification of the parking slot, it can then decide whether the vehicle is in a position and it can park safely or not. Range information is retrieved by ultrasonic sensors as time-of-flight (TOF) method is used to measure the distance between lateral sides of the vehicle and respective parking slots. As the driverless vehicle must follow the planned path for safe parking if the scheduled route would not be modified, the vehicle's initial position is

important. The distance information conveyed from TOF sensors will be used to judge whether the vehicle is located at the position, which would not require further control and its path correction. In this report, we will introduce the concept and simple pseudo-code for parking algorithm.

Software Design

Under the notion that software program would be designed under the hardware constraints of contemporary autonomous vehicle, our team mainly planned to utilized AVM system and machine-learning. Software program of this project consists of three parts. The first part is the code programmed on the RaspberryPi board that commands fisheye cameras to capture the surrounding environment and creates a birds-view image. The second part of the code implemented in the RaspberryPi board also processes the birds-view images to recognize parking slot type, angle, and its availability. The third part of the code locates its initial position, applies pre-determined path with respect to its position to the empty slot, and controls servo and DC motors for steering and propulsion.

In writing software for each part, our team referred to the latest articles and reports. Our goal was to find the most adequate studies to realize our ideas into a tangible output. As a result, the team held numerous discussions to brainstorm what sorts of algorithms would be required and searched for the closest published articles in AVM development and parking slot recognition. Also, as the aim of this project is to develop and validate the concept of autonomous parking for electric vehicle battery charging, our team occasionally relied on pseudo-code if it serves the project's purpose.

Around View Monitoring (AVM)

Around View Monitoring system, also called 'surround vision monitoring system,' provides a 360-degree view of the area surrounding the vehicle to its driver. Normally, there are four to six fish-eye cameras, each facing a different direction, mounted on the vehicle to capture the surrounding images. As each camera captures images at certain angle of view, the captured images are then processed and composited into a single spatial image of a top-down view. The top-down view, also called 'a bird-eye view,' effectively assists a driver to easily park his or her car into a parking spot with full attention to the surrounding environment.

In this project, we utilized AVM system to provide input images to recognize available parking slot. The input image of 360 degree surrounding bird-eye view is created in the following order: 1) camera calibration, 2) geometric alignment, and 3) composite view synthesis. We mainly referred to opensource codes in 'github.com' and relevant articles, which the reference are listed in appendix.

First, we need to identify which parameters for a RPi fisheye camera lens module should be calibrated. Intrinsic parameter of K (camera intrinsic matrix) consists of factors that define internal structure of a camera module. Such factors include focal length (fx, fy), and principal point (cx, cy). Extrinsic parameter D (distortion coefficients) is a transformation relationship between the frame of reference for camera and the frame of reference for world. Therefore, cameras connected to our RaspberryPi should be placed on a flat surface and be firmly fixated.

To retrieve K and D, our project utilizes well-defined samples chessboard images. We used cv2.findChessboardCorners() function to find the patterns of the chessboard, specifically by locating the corners of each square. Located corners are then compared with the given size of the chessboard. This is to find relationship between the retrieved 2D image (img_ points) and given 3D real-world information (obj_points). We finally use cv2.fisheye.calibrate() to find values of K and D.

```python
CHESSBOARD_SIZE = (6, 9) # from image


def calibrate(chessboard_path, show_chessboard=False):
    # Logical coordinates of chessboard corners
    obj_p = np.zeros((1, CHESSBOARD_SIZE[0] * CHESSBOARD_SIZE[1], 3), np.float32)
    obj_p[0, :, :2] = np.mgrid[0:CHESSBOARD_SIZE[0], 0:CHESSBOARD_SIZE[1]].T.reshape(-1, 2)

    obj_points = []     # 3d point in real world space
    img_points = []     # 2d points in image plane.

    # Iterate through all images in the folder
    image_list = os.listdir(chessboard_path)
    gray = None
    for image in image_list:
        img = cv2.imread(os.path.join(chessboard_path, image))
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        ret, corners = cv2.findChessboardCorners(gray, CHESSBOARD_SIZE,
                                                 cv2.CALIB_CB_ADAPTIVE_THRESH + cv2.CALIB_CB_FAST_CHECK + cv2.CALIB_CB_NORMALIZE_IMAGE)
        if ret:
            # Refining corners position with sub-pixels based algorithm
            obj_points.append(obj_p)
            cv2.cornerSubPix(gray, corners, (3, 3), (-1, -1),
                             (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.01))
            img_points.append(corners)
            print('Image ' + image + ' is valid for calibration')
            if show_chessboard:
                cv2.drawChessboardCorners(img, CHESSBOARD_SIZE, corners, ret)
                cv2.imwrite(os.path.join('fisheye_camera_undistortion/Chessboards_Corners', image), img)
```

```python
    k = np.zeros((3, 3))
    d = np.zeros((4, 1))
    dims = gray.shape[::-1]
    num_valid_img = len(obj_points)
    if num_valid_img > 0:
        rvecs = [np.zeros((1, 1, 3), dtype=np.float64) for _ in range(num_valid_img)]
        tvecs = [np.zeros((1, 1, 3), dtype=np.float64) for _ in range(num_valid_img)]
        rms, _, _, _, _ = cv2.fisheye.calibrate(obj_points, img_points, gray.shape[::-1], k, d, rvecs, tvecs,
                                                cv2.fisheye.CALIB_FIX_SKEW,
                                                (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 1e-6))
    print("Found " + str(num_valid_img) + " valid images for calibration")
    return k, d, dims
```

K and D constants after calibration are presented as the below.

$$K = \begin{bmatrix} 206.99838335 & 0 & 330.47416584 \\ 0 & 194.68300024 & 220.13338017 \\ 0 & 0 & 1 \end{bmatrix}$$

$$D = \begin{bmatrix} -0.3800582 & 1.14470312 & -1.67691534 & 0.81784743 \end{bmatrix}$$

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

width = 640; height = 480;

fx = 206.99838335; fy = 194.68300024;cx = 330.47416584; cy = 220.13338017;

dOne = -0.38010582; dTwo = 1.14470312;dThree = -1.67691534; dFour = 0.81784743;

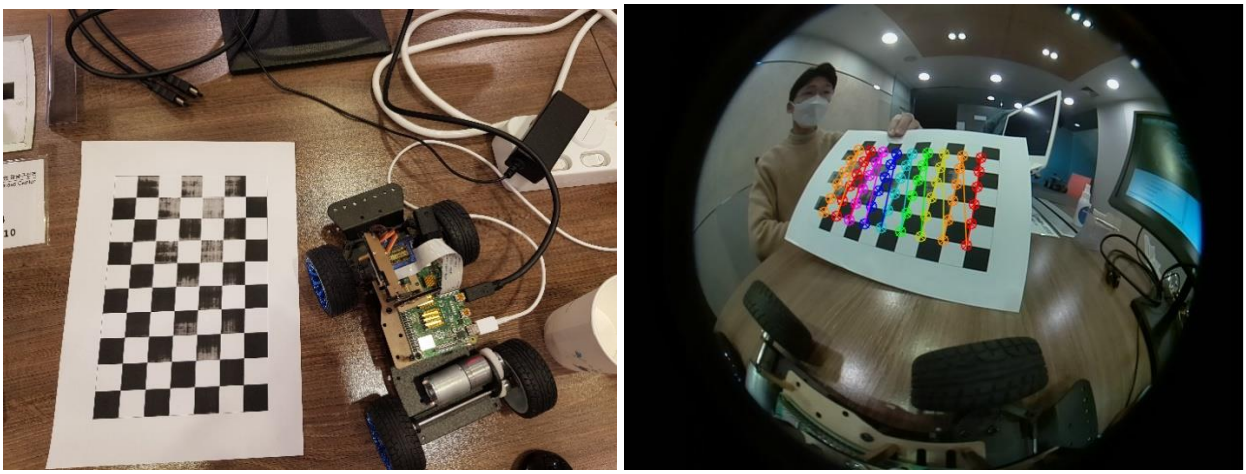


Figure 4. Camera Calibration

We then set ROI (region of interest) on the captured image by considering the nearby environment and omitting the rest. Afterward, we used cv2.initUndistortRectifyMap() and cv2.remap() to flatten the round distorted image into a flat one that assimilates what people see and perceive.

```
class UndistortFisheye:
    def __init__(self, cameraName, tune=False):
        self.tune = tune
        self.width  = 0
        self.height = 0
        self.fx = 0.0
        self.fy = 0.0
        self.cx = 0.0
        self.cy = 0.0
        self.dOne  = 0.0
        self.dTwo  = 0.0
        self.dThree = 0.0
        self.dFour  = 0.0

        regexVariable = r"(\w*)"
        regexValue = r"[-+]?\d*\.\d+|\d+"
        file = open("Parameters/K_D_Values_" + cameraName + ".txt", "r")
        lines = file.readlines()
        for line in lines:
            variable = re.findall(regexVariable, line)
            value = re.findall(regexValue, line)
            self.setMember(variable[0], value[0])

        self.DIM = (self.width, self.height)
        self.K = np.array([[self.fx,0.0,self.cx],
                           [0.0,self.fy,self.fy],
                           [0.0,0.0,1.0]])
        self.D = np.array([[self.dOne], [self.dTwo], [self.dThree], [self.dFour]])
```

```
def undistort(self, image):
    if self.tune:
        self.tuneDistortionVictor()
    map1, map2 = cv2.fisheye.initUndistortRectifyMap(self.K, self.D, np.eye(3), self.K, self.DIM, cv2.CV_16SC2)
    undistortedImage = cv2.remap(image, map1, map2, interpolation=cv2.INTER_LINEAR, borderMode=cv2.BORDER_CONSTANT)
    return undistortedImage
```

Figure 5. Transforming a Distorted Image into an Undistorted Image

Finally, our project transformed the converted undistorted image into a bird-view perspective. Through trial and errors, we found the optimal values for perspective transformation matrix M. The matrix is applied to transform the perspective from a camera-view to a bird-view.

```python
class BirdView:
    def __init__(self):
        self.__topLeft = []
        self.__topRight = []
        self.__bottomRight = []
        self.__bottomLeft = []
        self.__newDimensions = []

        self.__newWidth = None
        self.__newHeight = None
        self.__destinationDimensions = []
        self.__M = [] # Perspective Transform Matrix

    def setDimensions(self, topLeft, topRight, bottomRight, bottomLeft):
        self.__topLeft = np.array(topLeft).reshape(1, -1)
        self.__topRight = np.array(topRight).reshape(1, -1)
        self.__bottomRight = np.array(bottomRight).reshape(1, -1)
        self.__bottomLeft = np.array(bottomLeft).reshape(1, -1)
        self.__newDimensions = np.array([self.__topLeft, self.__topRight, self.__bottomRight, self.__bottomLeft], dtype="float32")

    def transfrom(self, image):
        self.__computeNewDimensions()
        self.__computeDestinationDimensions()
        self.__computePerspectiveTransformMatrix()

        birdView = cv2.warpPerspective(image, self.__M, (self.__newWidth, self.__newHeight))
        return birdView
```

```python
def __computeNewDimensions(self):
    # compute width of new image
    # width of new image is the longest of the distances between topLeft & topRight
    # or between bottomLeft & bottomRight corners
    topWidth = dist.cdist(self.__topLeft, self.__topRight, 'euclidean')
    bottomWidth = dist.cdist(self.__bottomLeft, self.__bottomRight, 'euclidean')
    self.__newWidth = max(int(topWidth), int(bottomWidth))

    # compute height of new image
    # height of new image is the longest of the distances between topLeft & bottomLeft
    # or between topRight or bottomRight corners
    leftHeight = dist.cdist(self.__topLeft, self.__bottomLeft, 'euclidean')
    rightHeight = dist.cdist(self.__topRight, self.__bottomRight, 'euclidean')
    self.__newHeight = max(int(leftHeight), int(rightHeight))

def __computeDestinationDimensions(self):
    topLeft = [0, 0]
    topRight = [self.__newWidth - 300, 0]
    bottmRight = [self.__newWidth - 300, self.__newHeight - 100]
    bottomLeft = [0, self.__newHeight - 100]
    self.__destinationDimensions = np.array([topLeft, topRight, bottmRight, bottomLeft], dtype="float32")

def __computePerspectiveTransformMatrix(self):
    self.__M = cv2.getPerspectiveTransform(self.__newDimensions, self.__destinationDimensions)
```

The following are the result of a bird-view transformation. The blackened areas are outside our region of interest, thus not considered. Note that this project did not consider synthesizing images in all four directions as image in one direction was enough for validation.

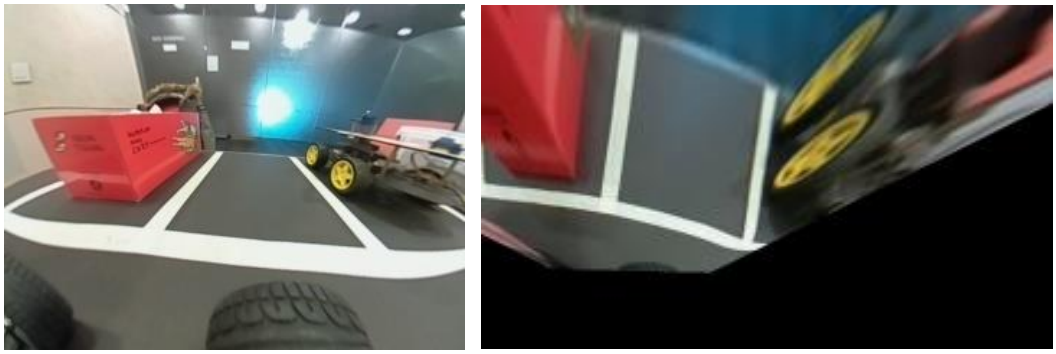


Figure 6. Transforming Perspectives into a Bird-Eye View

Context-Based Parking Slot Detection

After retrieving a bird-view image, the next step is to recognize where the parking slot is located and determine whether the slot is occupied or not. In finding the most updated algorithms, several discussions have preceded. Initially, our team, as with many other articles on image processing, searched for better ways to process parking slot images. Relevant articles include 'vps net' [Vacant Parking Slot Detection in the Around View Image Based on Deep Learning] or 'dmpr ps' [A novel approach for parking-slot detection using directional marking-point regression].

Yet, upon closer reflection on how people think when they try to park their cars, we soon realized that people have completely different understanding on objects they perceive. For instance, white lanes you see on highway are perceived as boundaries over which the car should not cross unless the driver wants to change the lane. When it comes to parking, however, drivers look for a rectangular space sizable enough for the car to fit in. Oftentimes the slot is drawn with white borders and these borders are identical to white lanes we see in a highway. Therefore, for the computer perceiving the identical white lines to determine whether it should command the vehicle to drive or park, our conclusion is that a context-based decision must be taken account of. In that regard, we found the latest article published in August 2020 that illustrates context-based parking slot detection. As the 'parking context recognizer 'looks for the availability of the parking slot and infer parking information, the algorithm is not confined by a set of image processing rules of the previous research but can identify diverse forms of parking slots.

Context-based parking slot detection consists of two stages. The first stage is called 'Parking Context Recognizer (PCR).' PCR operates to recognize the availability of a nearby parking slots. When AVM images of surrounding environment are passed as Input data, the backbone network called MobileNetV2 processes the data to infer types and rotation angles of detected parking lot. According to the article, MobileNetV2 network has shown the most reasonable performance with the fastest processing time and the smallest model size (In this project, we will not elaborate each network's performance). One feature of PCR is that it does not provide the type and slot angle unless it recognizes that the surrounding environment is available. This reduces the amount of calculation in actual parking, and make the model be suitable for an embedded system. PCR estimates four types of parking slots, namely 'parallel', 'perpendicular', 'diagonal,' and 'non-parking space'. After the classification is complete, PCR performs orientation regression of the detected parking slot. The angle ranges from -90 and + 90 degrees.

The second stage is called ‘Parking Slot Detection (PCD).’ At this stage, the algorithm locates the exact coordinates of the four vertices of the parking lot (quadrangle). Object detectors such as YOLOv3 is based, and rotated anchor box is created based on the estimated rotation angle. Rotated anchor box is that utilized to estimate the coordinates.

The following code, LaneDetector class, is an integrated module consisting vision algorithms for detecting lanes in autonomous parking.

```
class LaneDetector:
    def __init__(self, config):

        self.config = config

    def gaussian_blur(self, img: np.array, kernel_size: int):  # gaussian filter
        return cv2.GaussianBlur(img, (kernel_size, kernel_size), 0)

    def cannyEdge_img(self, img, min_thresh=70, max_thresh=210):
        return cv2.Canny(img, min_thresh, max_thresh)

    def weighted_img(self, img, initial_img, α=1, β=1.0, λ=0.0):  # overlap image
        return cv2.addWeighted(initial_img, α, img, β, λ)
```

pcr_detect method preprocesses a bird-view image by resizing and normalizing it as to be input for MobilinetV2. The method retruns parking lot type and angle as output. Note that real time basis regression requires used session to be cleared, so that it does not crash with a PSD model.

```
    def pcr_detect(self, img):

        parking_context_recognizer = get_model()

        img = cv2.resize(
            img,
            (
                self.config.CONTEXT_RECOGNIZER_NET["IMG_WIDTH"],
                self.config.CONTEXT_RECOGNIZER_NET["IMG_HEIGHT"],
            ),
        )
        img = img[np.newaxis, :]
        img = tf.image.per_image_standardization(img)
        result = parking_context_recognizer.predict(img, steps=1)

        type_predict, angle_predict = result
        tf.keras.backend.clear_session()

        type_predict = np.argmax(type_predict, axis=1)
        angle_predict = angle_predict * 180.0 - 90.0

        return type_predict, angle_predict
```

psd_detect method receives parking slot type and angle from PCR model as arguments. It also receives a bird-view image as img argument. The image is divided by 255 and normalized to a value between 0 and 1. After loading the fine-tuned model according to the parking slot type, PSD predicts the coordinates of the parking slot. Note that used session needs to be cleared no to conflict with PCR as PSD is in real time basis. The below test results show a successful parking slot detection when using this model with a sample parking lot picture.

```
def psd_detect(self, result, img):
    img = img / 255
    type_predict, angle_predict = result
    weight_path = "models/context_based/weight_psd/fine_tuned_type_" + str(type_predict[0])
    result, sess = detect_slot(angle_predict[0], weight_path, img)
    return result, sess
```

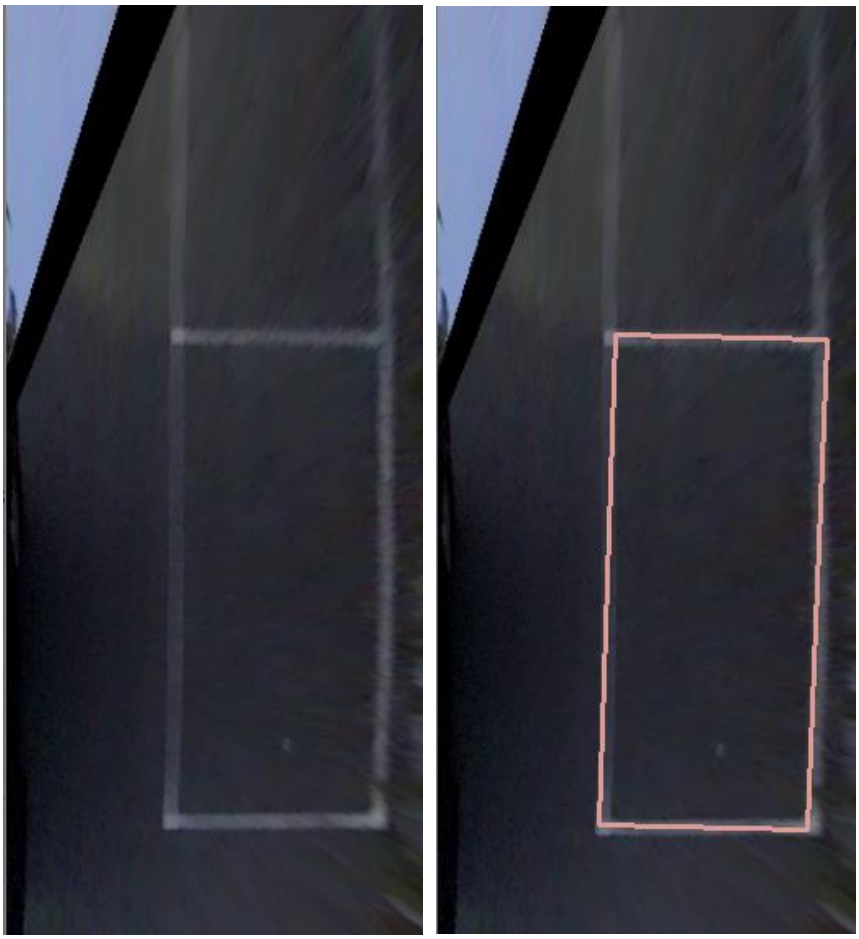


Figure 7. Recognizing an Available Parking Slot by **Context-Based Parking Slot Detection**

Parking Control

Class ‘Car’ implements the functionality in car control for driving and parking. Class-specific constant values of vehicle dimension are set. Parameterized values of parking environment such and control modules for lane detection and vehicle control are provided for methods to use.

```
class Car:
    def __init__(self, config, env, laneDetector, parkingControl):

        self.width = 10
        self wheel_base: int
        self.wheel_base = 20
        self.front_overhang = 5
        self.rear_overhang = 5
        self.parkingMode = parkingControl(self, env)

        self.start_pos_x = 0
        self.config = config
        self.env = env

        self.laneDetector = laneDetector(config)
        self.curMode = None
```

Referring to one of the published algorithms in autonomous parking, our team summarized it as below and wrote a pseudo code for it.

1. Drive alongside the parking spots.
2. Vacant Parking Slot is detected in the Around View Image.
3. Vehicle stop and path planning for parking.
    1) find parking category (Backward parking or Front Parking or Parallel Parking)
    2) path planning (1st arc + 2nd arc + linear) and steering angle control (-θmax , +θmax, 0)
4. Vehicle move to parking start position.
5. Parking execution (from start position to end position)

Parking Control Pseudo Code

$Input:\ Around\ View\ Image$

$Algorithm$

$Initialize\ Car\ Position\ X, Velocity\ V\ , Steering\ Angle\ \alpha, done\ False$

$While\ not\ done:$

$Drive\ Along\ the\ lane$

$If\ Detect\ Parking\ Lot(Around\ View\ Image):$

$\theta_l, \theta_r \leftarrow Calculate\ Maximum\ Steering\ Angle(X, V, \alpha)$

$\theta_{steer} \leftarrow \frac{\theta_l + \theta_r}{2}$

$radius_{min} \leftarrow \frac{wheel\ base}{tan\theta_{steer}}$

$If\ no\ collision(X, \theta_{steer}, radius_{min}):$

$StartParking$

$done \leftarrow True$

To elaborate, minimum rotating radius(r) for the vehicle is considered for path planning. For our project, the vehicle specification presents its minimum vehicle turning radius as 450 mm. Referring to Ackerman steering geometry, we could find from various articles that the turning radius r is determined by the value of steering angle and vehicle wheelbase. That is, when θl and θr are at their maximum steer angle values, θsteer is at its maximum and r is at its minimum.

$$r = \frac{wheelbase}{\tan\theta_{steer}}, \quad \theta_{steer} = \frac{\theta_l + \theta_r}{2}$$

ParkingControl class is a pseudo code for controlling parking. StartParking method calculates ‘Dubins Curve’ by utilizing geometrical information between the car’s current location and parking slot location. The vehicle will then rear park, following the calculated path.

```python
class ParkingControl:
    def __init__(self, car, parkginLot):

        self.car = car
        self.parkingLot = parkginLot

    def StartParking(self, cur_x: float, cur_y: float) -> float:

        min_r = self.parkingLot.wall_offset + self.car.position.y  # min radius
        self.car.stop()

        target_pos_x = abs(self.car.position.x - self.env.k1.x) + min_r
        self.car.move_x(target_pos_x)  # move by x-axis

        slope = (self.car.width + min_r) / self.car.wheel_base  # calculate steering angle
        beta = math.atan(slope)

        self.car.steer(beta)
        self.car.drive(-100, self.endCondition)  # go backwards until close to the wall

    def endCondition(self):
        """주차 종료 조건"""
        if abs(self.car.position.y - self.env.space) < self.env.limit:
            return True
        else:
            return False
```

Hardware Design and System Integration

The overall hardware wiring schematic for the entire system can be seen below. The wiring integrates one servo motor that steers the front wheels, one DC motor that drives the rear wheels, and two ultrasonic sensors located in the sides of the vehicle to detect nearby object to avoid collision. Servo motor is connected to PCA9685 driver, and DC motor is connected to TB6612FNG driver.

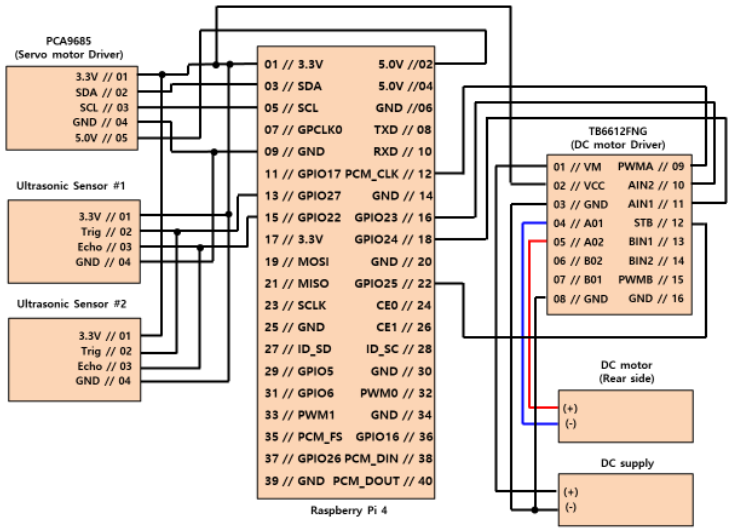


Figure 8. Schematic of Overall System GPIO I/O Pins

Because the output voltage of DC motor is 12V, it should be connected to the power source of the equivalent voltage. Due to the limited available space on the vehicle, our team added second floor on top of the car to place components and a battery. We put the battery in the middle of the first floor for a more stable center of gravity. We placed RaspberryPi, wiring breadboards, and a fisheye camera on the second floor of the vehicle. In the future, we plan to employ XL6009 step-up converter to simplify power source to acquire more space.

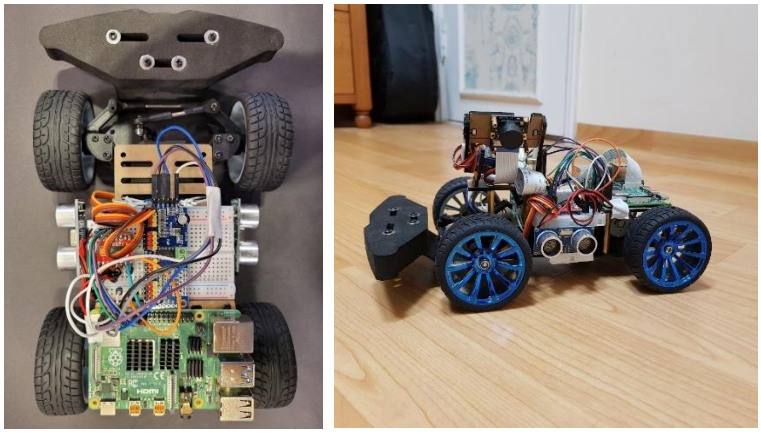


Figure 9. Hardware Assembly

Testing and Results

Hardware Testing

Major components for the autonomous parking vehicle are DC motor for propulsion, servo motor for steering, ultrasonic sensor for collision detection, and fisheye cameras for image detection. The following are the test code utilizing some methods such as GPIO from RaspberryPi system. We mainly test whether each component functions properly. Test results were all successful.

DC/Servo Motor

```
#Servo motor control
import RPi.GPIO as GPIO
import time import sleep
GPIO.setmode(GPIO.BOARD)
GPIO.setup(12, GPIO.OUT)
p = GPIO.PWM(12,50)
p.start(0)
p.ChangeDutyCycle(3)
sleep(1)
p.ChangeDutyCycle(12)
sleep(1)
p.ChangeDutyCycle(7.5)
sleep(1)

while(1):
    val = float(raw_input(“input(3~7.5~12) = “))
    if val == -1 : break
    p.ChangeDutyCycle(val)
p.stop()
GPIO.cleanup()

#DC motor 17ontrol
from time import sleep   #Import sleep from time
import Rpi.GPIO as GPIO  # Import Standard GPIO Module
GPIO.setmode(GPIO.BOARD)
GPIO.setwarning(False);
#PWM Frequency
pwmFreq = 100
#Setup pins for motor controller
GPIO.setup(12, GPIO.OUT) #PWMA
GPIO.setup(18, GPIO.OUT) #AIN2
GPIO.setup(16, GPIO.OUT) #AIN1
```

```
GPIO.setup(22, GPIO.OUT) #STBY
pwma = GPIO.PWM(12, pwmFreq)
pwma.start(100)
#Functions
def forward(spd):
    runMotor(0, spd, 0)
def reverse(spd):
    runMotor(0, spd, 1)
def runMotor(motor, spd, direction):
    GPIO.output(22, GPIO.HIGH);
    in1 = GPIO.HIGH
    in2 = GPIO.LOW
    if(direction==1):
        in1 = GPIO.LOW
        in2 = GPIO.HIGH
    if(motor==0):
        GPIO.output(16,in1)
        GPIO.output(18,in2)
        pwma.ChangeDutyCycle(spd)
def motorStop():
    GPIO.output(22, GPIO.LOW)
##Main##
def main(args=None):
    while True:
        forward(50)  # run motor forward
        sleep(2)     # sleep for 2 sec
        motorStop()  # stop motor
        sleep(.25)   # delay between motor runs
        reverse(50)  # run motor in reverse
        sleep(2)     # sleep for 2 sec
```

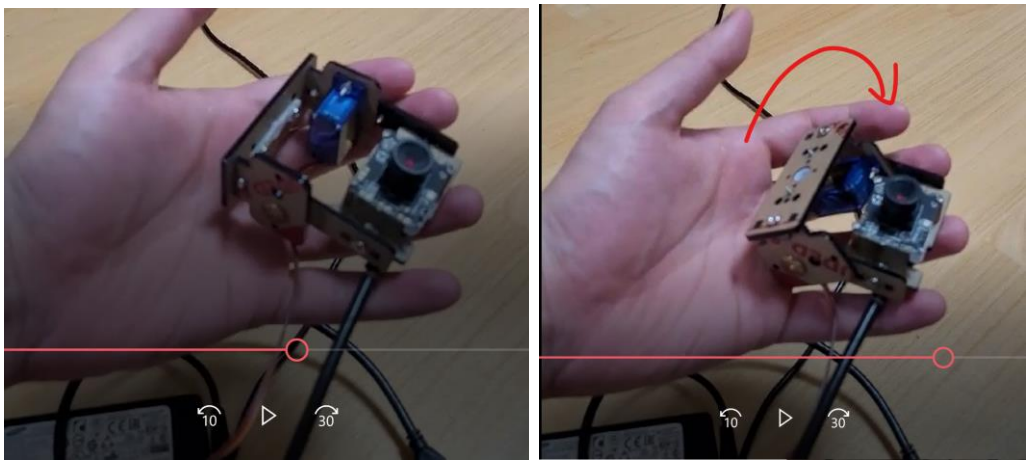


Figure 10. Successful Test Results of Servo Motor Actuation

Ultrasonic Sensor Testing

```
#Ultrasonic sensor control

import Rpi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)

trig = 13
echo = 19

GPIO.setup(trig, GPIO.OUT)
GPIO.setup(echo, GPIO.IN)

try :
   while True :
        GPIO.output(trig, False)
        time.sleep(0.5)

        GPIO.output(trig, True)
        time.sleep(0.00001)
        GPIO.output(trig, False)

     while GPIO.input(echo) == 0 :
        pulse_start = time.time()
     while GPIO.input(echo) == 1 :
        pulse_end = time.time()

     pulse_duration = pulse_end - puse_start
     distance = pulse_duration * 17000
     distance = round(distance, 2)

     print ("Distance : ", distance, "cm")


except :
    GPIO.cleanup()
```

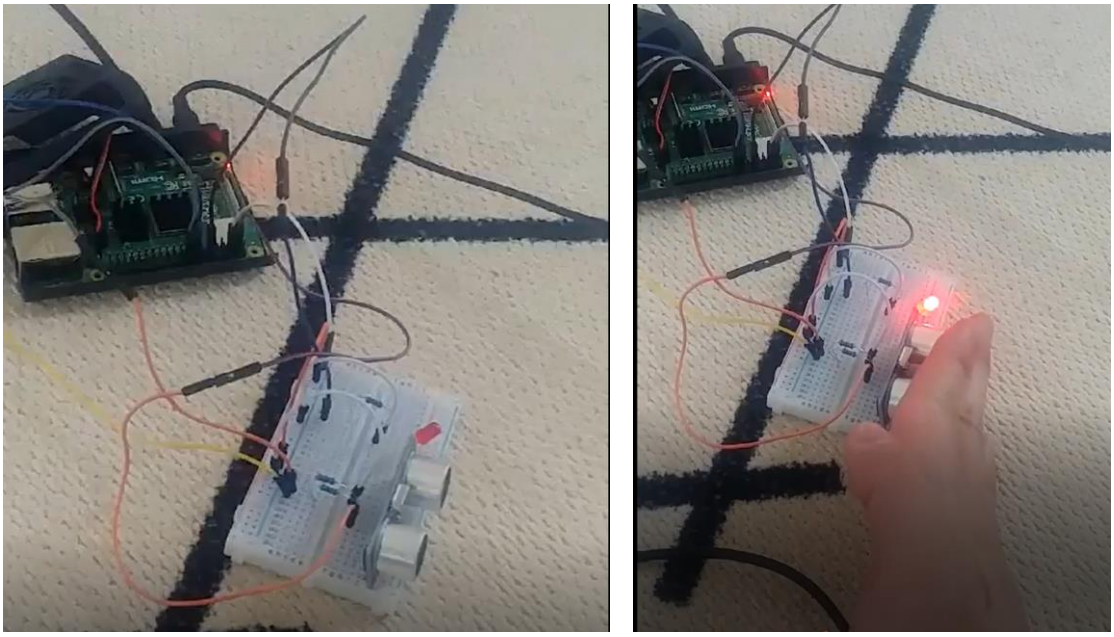


Figure 11. Successful Test Results of an Ultrasonic Sensor

Test Environment Settings and Results

In the end, our vehicle performed as planned and met the goals we expected. In every step of our software design, it presented successful results with sample data including chessboard squares and pictures of random parking lots. Regarding our hardware components, we successfully implemented the components to the RaspberryPi system and functioned them properly. In case of parking control, our project gave less weight to it and instead wrote pseudo-code to complete our conceptualization.

After conducting validation for each component and software, we integrated into a vehicle level and created a testing environment, which endeavors to abstract the real-world. This is because we could not test the concept on real cars, and thus the test could not be conducted on a real environment. We drew lanes with white-colored tape, placed obstacles inside created parking slots to imitate cars occupying adjacent slots, and used black piece of paper to resemble typical roads.

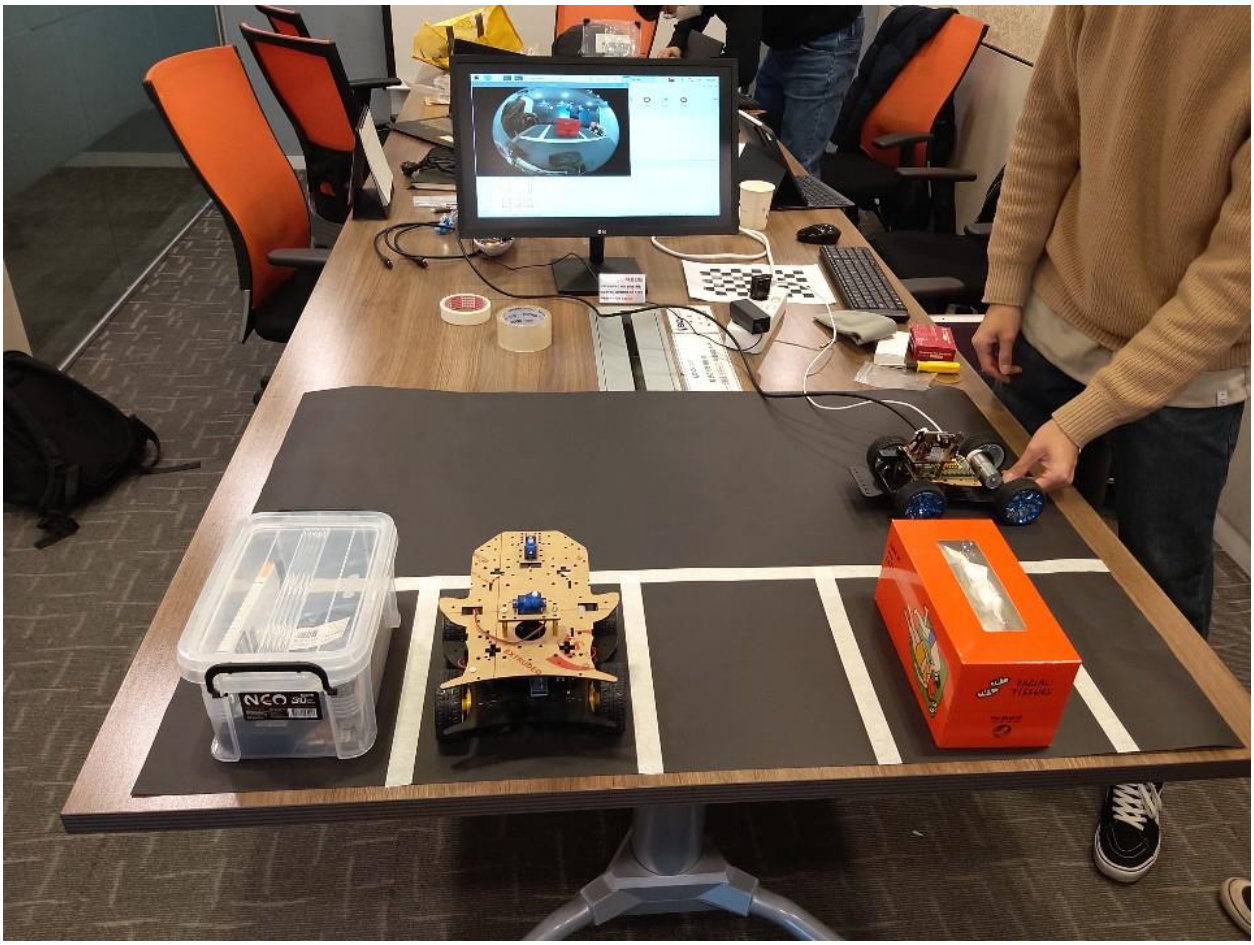


Figure 12. Test Environment Settings

After conducting validation for each component and software, we integrated into a vehicle level and created a testing environment, which endeavors to abstract the real-world. This is because we could not test the concept on real cars, and thus the test could not be conducted on a real environment. We drew lanes with white-colored tape, placed obstacles inside created parking slots to imitate cars occupying adjacent slots, and used black piece of paper to resemble typical roads.

The following pictures are sequence of the test conducted. With the embedded software running on a vehicle, one of our members slowly moved the vehicle forward and captured pictures of the surrounding environment. It shows that the image is well displayed on a screen, which would then be converted into a bird-eye view.

However, it can be seen from the lower image that the vehicle fails to recognize an unoccupied parking slot from the given bird-eye view (Figure 13). Proceeding investigation, our team tested Hough Transform to check whether there is any problem in image processing, but a success in the result showed implied other causes for the failure (Figure 14). Finally, our team identified the root cause as dissimilarity between drawn white lanes and real-world parking lanes. Lanes in the test settings were too wide in their width, and the ration of the quadrangle was also different.

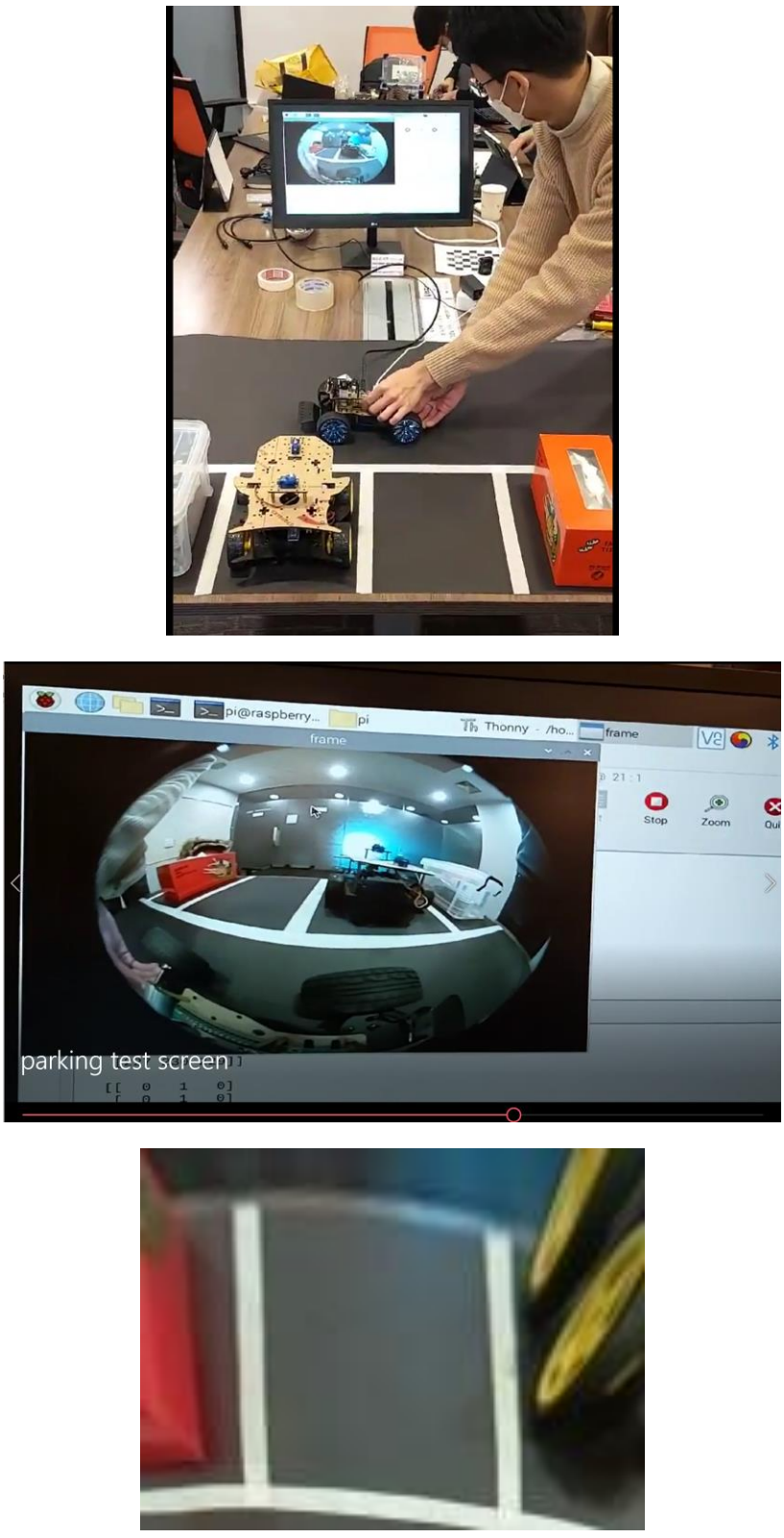


Figure 13. Initial Test Failure due to Defects in Environment Settings

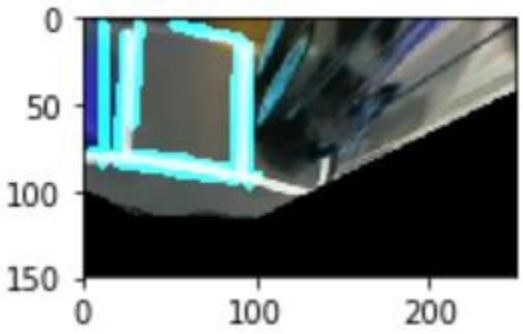


Figure 14. Success Detection by Hough Transform

Therefore, our team corrected the bird-view image to have parking slots like the real-world ones. The width of the lanes and the ratio of the quadrangle were corrected. Then, our team found that our autonomous parking system successfully recognized a vacant parking slot (Figure 15). Note that the blue lines are the detected parking slot.

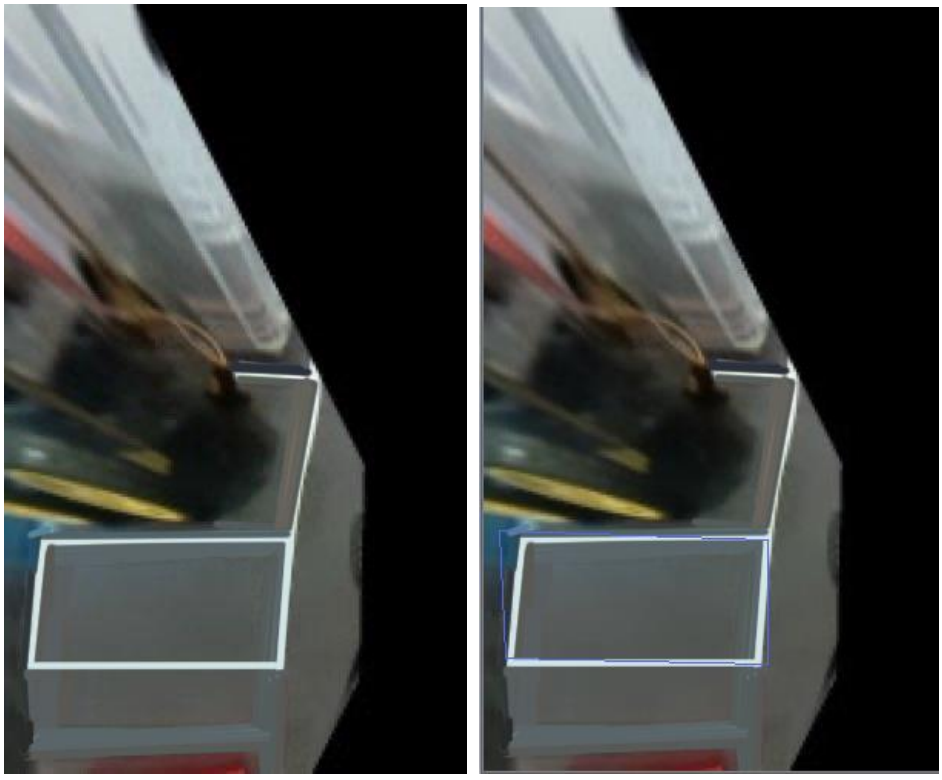


Figure 15. Success Detection by **Context-Based Parking Slot Detection**

Future Works

Autonomous Battery Charging Service

Electric Vehicles ranging from Hybrid, Plug-in hybrid (PHEV) to pure electric cars (BEV) require charging stations at a rapidly increasing rate. BEV and PHEV sales ratio has increased from 2.3% in 2019 to 16% in 2025, 16%, respectively, an approximate 7-fold increase. As illustrated in the figure below, in 2019 the total number of BEVs sold is 1.68 million cars and that of PHEVs sold is 0.58 million. It is estimated that in 2025, the total number of BEVs sold would be 11.47 million and that of PHEVs sold would be 0.51 million. According to Bloomberg New Energy Finance <2020 EV Outlook>, it is estimated that 58% of total passenger cars will be xEVs in 2040.

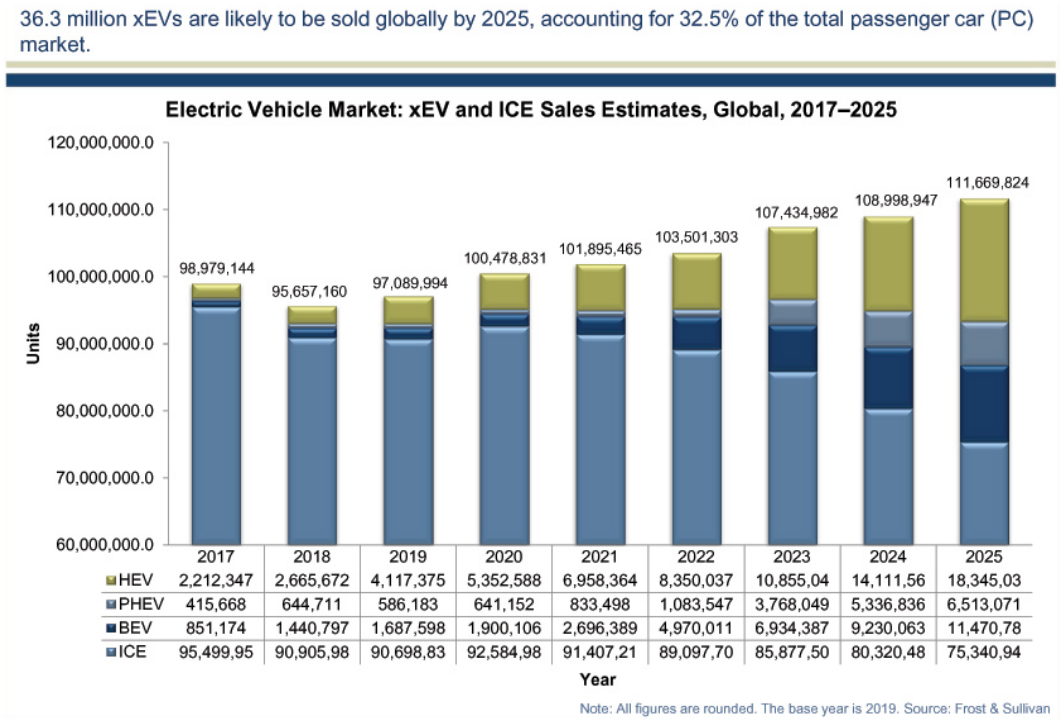


Figure 16. Electric Vehicle Market 2017 - 2025

In addition, International Energy Agency (IEA) outlooks that in 2030, more than 250 million vehicles would need their batteries to be charged.

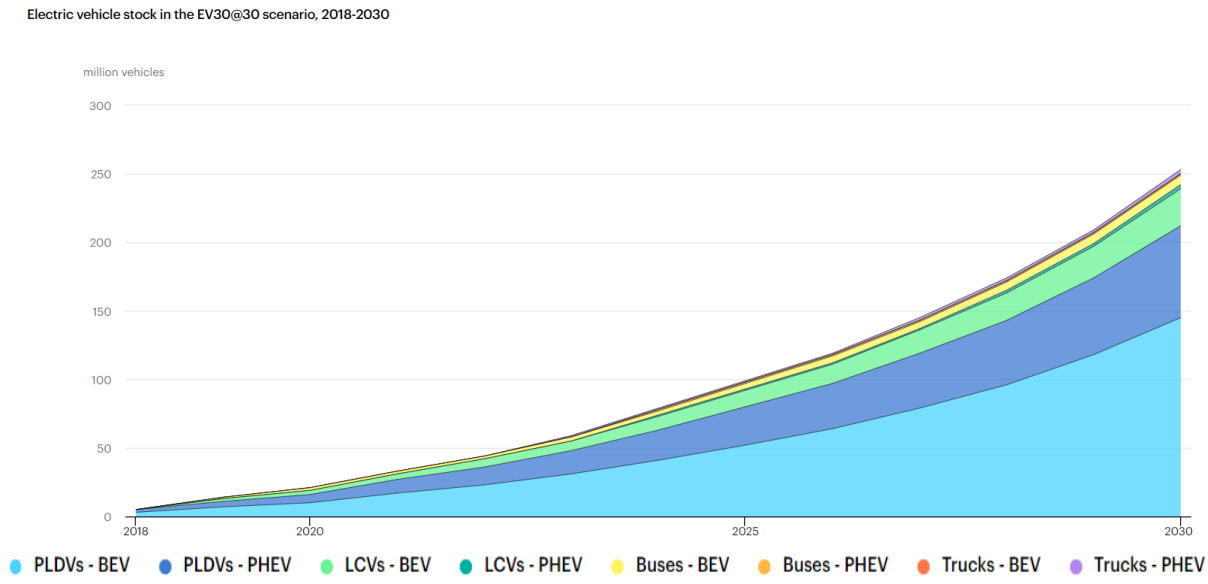


Figure 17. IEA Report – Global EV Outlook 2020

The below table describes the attributes of battery charging mode for contemporary electric vehicles. Electric vehicles of today provide two types of charging mode: a quick charge and a standby charge. Recently, Tesla's 'model Y' can charge its battery from SOC10% to 80% in 22 minutes. This is a noticeable difference compared to other OEMs who provide quick charge rate of 30 ~ 40 minutes from roughly 10% to 78%. However, 22 minutes of fast charge is still far from user's experience in fueling gasoline to Internal Combustion Engine (ICE) vehicles, which would normally take about 3 ~ 5 minutes to fill the empty tank. Moreover, fast-charge is not recommended for daily uses as it significantly reduces battery life cycles. With silicon oxide (SiO) included in battery anode for fast charge improvement, it is roughly anticipated that battery life cycle reduces to 80% within 8 years. Thus, some car manufacturers are devising ways to limit the number of fast-charge by the vehicle control system.

Charging Mode	Charging Time	Battery Life Cycle
Quick Charge	Tesla Model Y: 22 min (SOC 10-80%) OEM average: 30 ~ 40min	Life cycle decreases rapidly
Standby(normal) Charge	Average 6hrs	Life cycle decreases slowly

Figure 18. Difference in Battery Charging Mode and Effect

In contrast, standby charge mode would need approximately 6 hours for the battery to be charged. For regular customers, this mode is useful when they park for a long time in a charging station or when they plug the vehicle inside their garage. Battery life cycle decrease is minimized on standby

charge mode. However, several problems lie on such charge mode. First, the longer charge time is most suitable for people who can manage to own garages at home. For urban dwellers who live in apartments, condominium, or housings without enough space for garage, it is difficult to install a private charger to plug the car overnight. Second, when it comes to a charging station in a public area, 6 hours of charging time bounds the driver to be back to the station to unplug the charger and be responsible to empty the charging slot available for others. If the driver fails to be back to the station for some reason, then this will increase the queuing time of charging for others. Third, such constraint will likely discourage people to charge their cars during office hours. This is because moving back and forth from office to charging station will take much time during working hours. Less people will be able to charge their cars during office hours, but more will charge batteries during rush hour and cause traffics in occupying charging stations. Finally, such scenario mentioned before will less likely maximize the profit of charging station business. For charging stations, running the charger at every turn will likely increase the revenue.

As electric vehicles are becoming more popularized, congestion in battery charging will likely increase mainly due to the mismatch between charging time in demand and charging time in supply. Therefore, our project aims to utilize autonomous parking function as the main means to deliver driver-less cars into a charging station slot and support the charging activity via a robot arm. Also, this service will be provided through mobile application or web service, where users can choose adjacent unoccupied charging slots, while charging station business can increase their profits by registering their stations on this platform.

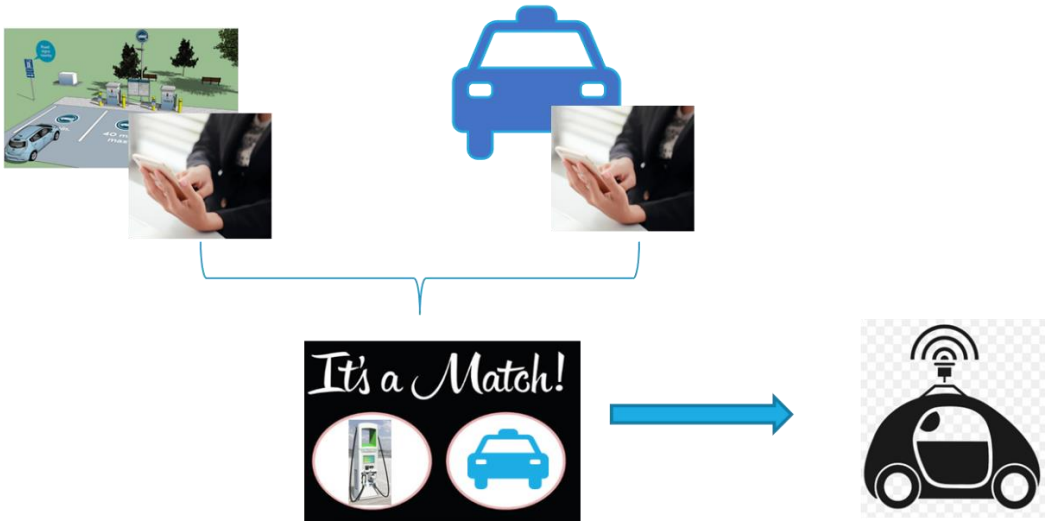
"Autonomous Charging Service" is a membership solution that 1) matches car drivers with charging stations, and 2) assists self-parking of an autonomous vehicle through solutions such as a battery charger plugging robot arm.

The following sequence is how 'Autonomous Charging Service' is performed.

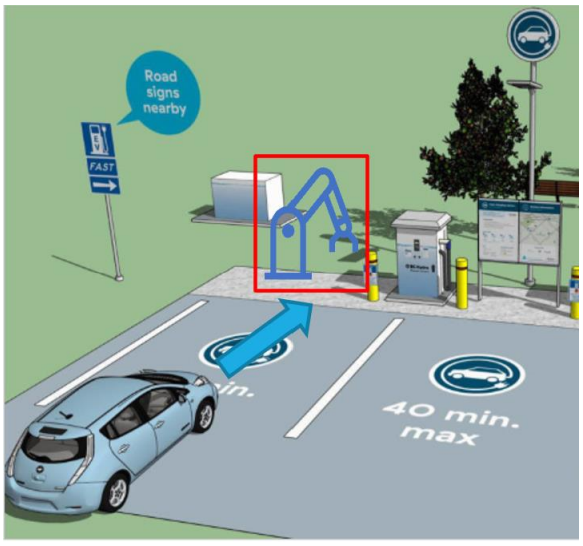
- Initial state: Car is parked (Driver is not in the vehicle)
- Find & Match a nearby available charging station owner
- Start an Autonomous Drive (driverless) to a nearby matched charging station
- Start Auto Charging (robot arm plugs the charger into the vehicle)
- Return Drive to the initial location or to a designated parking lot by a user.

Thus, as illustrated in the below picture, drivers can send their autonomous vehicles to a vacant charging station during their office hours and charge the vehicle on standby mode without concern to visit the station and return the car. Robot arm will automatically pick up the charger and plug it

to the parked vehicle. After charging is complete, the robot will unplug the charger and the vehicle will return to its initial position or a designated parking lot the user has assigned.



Matching charging stations with autonomous electric vehicles



Robot arm plugging a charger to a self-parked car

The following is a lean canvas for our project as a summary of business idea mentioned above. Further investigation in revenue stream and cost structure are required to develop the idea into a business plan. As our concept service is to connect charging station business with autonomous cars, it is reasonable to include development and maintenance cost of the application service in the cost structure. In case of robot arm, this can be included or if the current cost of the robot is too expensive, then cheaper labors can replace the robot until the cost of robotics become lower.

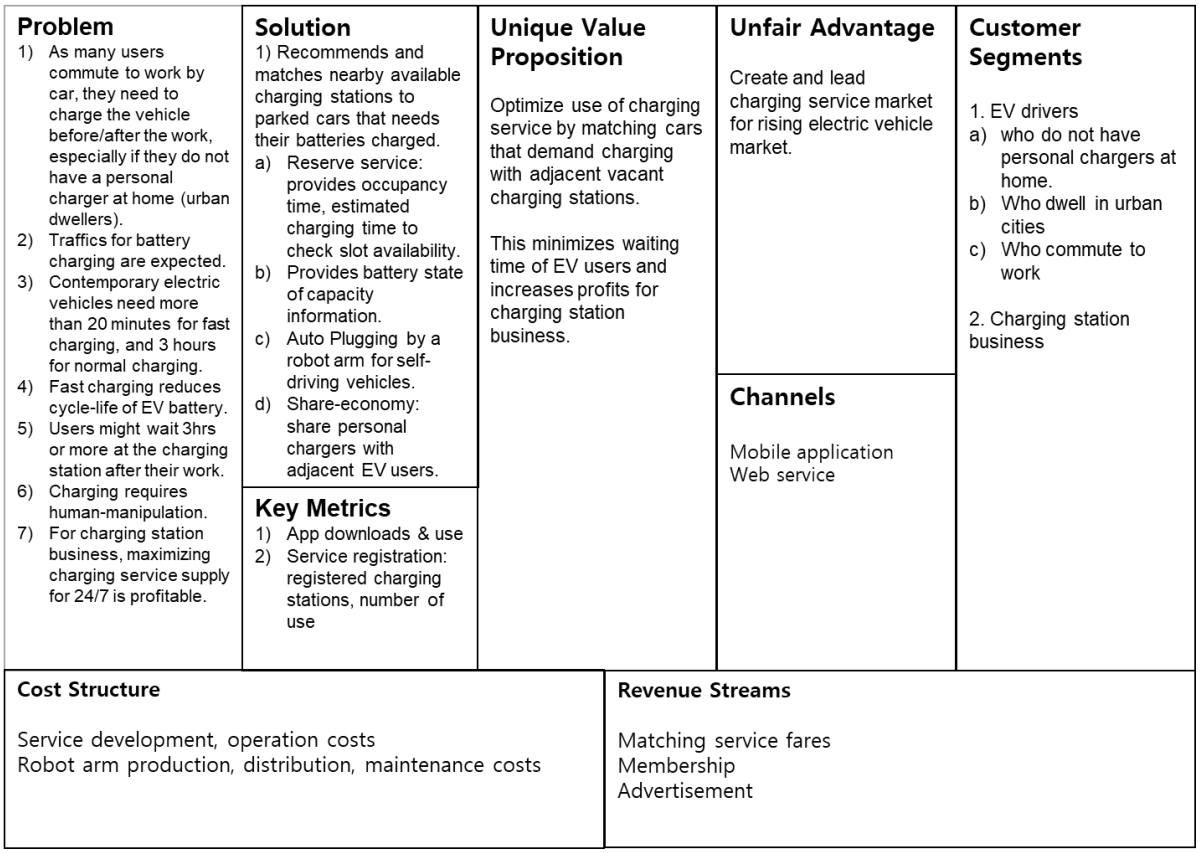


Figure 19. Autonomous Charging Business Plan Lean Canvas

Battery Charger Plugging Robot Manipulator

Figure 20. A Problem in Autonomous Charging

One of the bottlenecks for autonomous battery charging service is finding ways to automate plugging process. It would be uneconomical to change the established charging infrastructure to one with automated interface. Instead, other ways of plugging a battery charger into a self-driving car can be thought of. One way is to hire part-time people. This could also work in a sharing-economy form as it is now commonly performed among services such as 'Uber drivers.'

Another way is to implement a simple manipulator that can recognize where the charger should be plugged into, and to grasp the charger for manipulation. Several conditions for economic robot manipulator are discussed as the following.

- Safety
- Cost competitiveness
- Simple design
- Standardized components for procurement and reducing cost
- Deep Learning
- Identify vehicle models and their plug locations

In the project, we briefly reviewed the status of robot arms distributed in the market. While consumer robots present considerable potential for versatile applications, their prices are beyond everyday reach, hampering distribution to the public. Industrial robots are standardized to a great extent, but their focus on manufacturing confines them to stay within factory settings. It is this kind of gap which has motivated me to explore ways to utilize established industrial robotics resources to deliver professional service robots. Our team has found some of the candidates that utilizes

simple linear actuation but controlled by deep-learning algorithms for safe and precise actuation. For instance, ‘hello robot’ presents a new kind of mobile manipulator that can interact within human environment with its low mass and contact-sensitive body. The robot is designed for household affairs at home, but this design has potential to be developed into a professional service robot. Inspired with the introduced robot, our team will continue to devise the most suitable robot manipulator for our business idea.

Figure 20. A New Robot Manipulator from ‘Hello-Robot’ Company

Conclusion

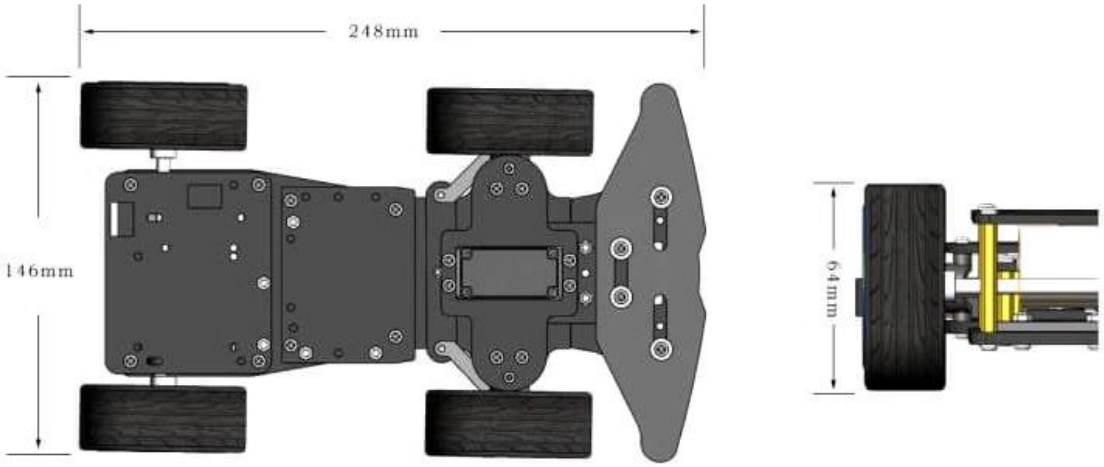
Professor Jeremy Wyatt presents the three waves of robotics during TEDx talk in 2016 (Wyatt 2016). The first wave of robotics is called ‘the structured manipulation,’ mainly referring to industrial robots confined in factory settings. These robots repeatedly perform precise controlled movement between two or three positions by exactly following the written programs. With no capacity for autonomy, however, such robots cannot deal with uncertainty as recognizing what the external environments are alike is not feasible. The second wave of robotics is called ‘the unstructured mobility.’ Being able to cope with uncertainties, robots are now freed from the cages and be able to mingle with our daily lives. To be more specific, probabilistic artificial intelligence enables planning with perceived data, which otherwise would be insufficient under previous control methods of industrial robots. Machine learning helps robots combine prior beliefs about the world and updated evidence of unreliable data. Thus, robots such as self-driving vehicles that perceives the real world have become available, and they are still rapidly developing even at this moment as more drivers are riding the cars and gathering much data to further develop AI algorithms. Finally, the third wave is ‘the unstructured manipulation,’ when robots can make autonomous judgments in manipulating objects so natural that they are virtually no different from how people manipulate objects.

This project reflected the prospect of robotics presented by Professor Jeremy Wyatt. in conceptualizing our business idea, we believed that the progress in robotics is important factor to be taken accounted for the business realization. In this regard, our team first addressed the current development status in autonomous parking. Tracking the latest articles on autonomous parking, our team realized that the technology has not been complete yet. Algorithms for autonomous parking requires more development, and other supplementary measures might be needed such as local map for mapping. As GPS does not function in underground parking lots or inside buildings, discovering means to assist autonomous vehicles in mapping might be another business opportunity. In fact, the resolution of GPS localization is not high enough to exactly locate a charging station booth, implying much room for improvement.

In addition, the advent of robot manipulator from ‘Hello-Robot’ is an indication that manipulators are under development in consideration of actual uses in our daily lives. For our conceptualization, a manipulator would have to simple and small enough to be placed in a small space between cars parked adjacent. On top of that, one must always take account of the cost and find ways to minimize it for business realization.

Appendix

Vehicle Frame Specification



- Vehicle weight: 700g
- Long * Width * Height: 235 * 146.8 * 64 [mm]
- Minimum ground distance: 16mm
- Minimum turning radius: 45cm

Member Contributions

Bruce KwangKyun Kim generated the idea of autonomous charging business. Bruce suggested how the business can start from a mobile service, a parking assistance and to a robot manipulation in the current autonomous electric vehicle market in relation to the charging infrastructure. He was also responsible as a product manager to lead the group during the entire conceptualization. IlGyu Cho mainly conducted research and investigation in mobile service market to verify and extend the idea's marketability. He presented the recent progress in robot manipulation by introducing 'Hello-Robot' to the team. He also carefully managed resources spent in the project. Limaries HanDong Lim was mainly responsible for the software development. He programmed the entire software structure and focused on 'Context-Based Parking Slot Detection,' and 'Parking Control'. Limaries also managed the entire progress of the work on his GitHub. Kunhee Han was responsible for both software and hardware development. He programed AVM algorithm and RPI implementation. Also, he assisted to design our team's model car for camera implementation, and managed test environment settings. HoYong Park was mainly responsible for hardware development and RPI implementation. He wired all the connections for the RPi with all hardware components such as the servo/DC motors, motor drivers, and ultrasonic sensors. The final overall system assembly and debugging required the combined efforts of all team members.

References

Around View Monitoring (AVM)

https://github.com/Nebula4869/fisheye_camera_undistortion
https://github.com/Ahid-Naif/Around-View-Monitoring-AVM

Surround view camera system for ADAS on TI's TDAx SoCs / Vikram V. Appia, H. Hariyani, S. Sivasankaran, S. Liu, K. Chitnis, M. Müller, Umit Batur, G. Agarwal/ 2015

Automatic Parking Space Detection and Tracking for Underground and Indoor Environments/Jae Kyu Suhr and Ho Gi Jung/2016

Recognition of Driving and Parking Lines for Intelligent Vehicles Using Bird Eye View System/ Hyun Soo Bae and Suk Gyu Lee/ 2018

Fully-automatic Recognition of Various Parking Slot Markings in Around View Monitor (AVM) Image Sequences/ Jae Kyu Suhr and Ho Gi Jung/ 2012

Parking Slot Markings Recognition for Automatic Parking Assist System/ Ho Gi Jung, Dong Suk Kim, Pal Joo Yoon, Jaihie Kim/ 2006

Context-Based Parking Slot Detection

https://github.com/dohoseok/context-based-parking-slot-detect

DO, Hoseok; CHOI, Jin Young. Context-Based Parking Slot Detection With a Realistic Dataset. *IEEE Access*, 2020, 8: 171551-171559.

Huang J, Zhang L, Shen Y, Zhang H, Zhao S, Yang Y. DMPR-PS: A novel approach for parking-slot detection using directional marking-point regression. In2019 IEEE International Conference on Multimedia and Expo (ICME) 2019 Jul 8 (pp. 212-217). IEEE.

Zhang L, Huang J, Li X, Xiong L. Vision-based parking-slot detection: A DCNN-based approach and a large-scale benchmark dataset. IEEE Transactions on Image Processing. 2018 Jul 18;27(11):5350-64.

Parking Control

A control system for autonomous vehicle valet parking (ETRI, 2013).
https://github.com/jovanduy/AutonomousParking

Others

https://www.tesla.com/

https://www.youtube.com/c/Dronebotworkshop1

Frost& Sullivan (2020.5)

IEA Report – Global EV Outlook 2020

https://hello-robot.com/