DESIGN OF AN UNMANNED GROUND VEHICLE CAPABLE OF AUTONOMOUS NAVIGATION

In partial fulfillment of the requirements for

Final M.S. Project, Autumn 2016

By:

Student Name
Master of Science in Electrical and Computer Engineering
The Ohio State University, Columbus

Under the Guidance of

Dr. LISA FIORENTINI
Assistant Professor
Department of Electrical and Computer Engineering
The Ohio State University, Columbus
# INDEX TO CHAPTERS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>TOPICS</th>
<th>PAGE NO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ABSTRACT</td>
<td>3</td>
</tr>
<tr>
<td>CHAPTER 1</td>
<td>BACKGROUND</td>
<td>4-5</td>
</tr>
<tr>
<td>CHAPTER 2</td>
<td>THE RASPBERRY PI</td>
<td>6-11</td>
</tr>
<tr>
<td>CHAPTER 3</td>
<td>MATLAB AND RASPBERRY PI</td>
<td>12-13</td>
</tr>
<tr>
<td>CHAPTER 4</td>
<td>SAMPLE CODES</td>
<td>14-18</td>
</tr>
<tr>
<td>CHAPTER 5</td>
<td>ENCODER DOCUMENTATION</td>
<td>19-21</td>
</tr>
<tr>
<td>CHAPTER 6</td>
<td>THE PROXIMITY SENSOR</td>
<td>22-23</td>
</tr>
<tr>
<td>CHAPTER 7</td>
<td>PWM GENERATION SYSTEM</td>
<td>24-28</td>
</tr>
<tr>
<td>CHAPTER 8</td>
<td>SYSTEM SIMULATION</td>
<td>30-42</td>
</tr>
<tr>
<td>CHAPTER 9</td>
<td>VALIDATION OF THE ENCODERS AND POSITION ESTIMATION ALGORITHM</td>
<td>43-45</td>
</tr>
<tr>
<td>CHAPTER 10</td>
<td>RESULTS AND CONCLUSION</td>
<td>46-48</td>
</tr>
<tr>
<td>CHAPTER 11</td>
<td>REFERENCES</td>
<td>49</td>
</tr>
</tbody>
</table>
ABSTRACT

This project is a multidisciplinary initiative with an objective to develop a scalable robotic platform of ground and aerial robotic swarms that are capable of tracking an object in an unknown environment. Conventionally, most robotic platforms perform tasks independently and are not designed to perform tasks in coordination with other robotic agents, thereby limiting their effectiveness in missions with critical time constraints. In contrast, the robotic platform developed in this project comprises of autonomous robots that are deployed as part of a large network of swarm robots. The ground robots receive mission objectives either from an operator or from a supervisory robot like an unmanned aerial vehicle (UAV). After receiving its mission objectives, the robot uses its own embedded intelligence to complete the mission in coordination with other robots. The robotic platform comprises of different subsystems that facilitate the control, coordination and communication between internal subsystems and other robots in the swarm network. Each robot consists of a chassis, sensor suite, drive unit, battery management system, navigation system, master control unit and a communication link. The robot is built on a differential wheel drive chassis system that supports locomotion in rugged terrains. The sensor suite of the robot enables it to develop a sense of situational awareness by measuring a range of physical parameters. It further provides a 360-degree view of the surroundings with limited hardware by using novel mounting strategies. The drive unit interfaces with the high torque motors and controls the speed and direction of the robot, while the battery management system ensures that all the subsystems receive the required power to remain operational. The master control unit acts as the brain of the robot and takes local control action to ensure that the mission objectives are completed most reliably. The robot has an inertial navigation system that performs position estimation, path planning and trajectory control. The communication link for the robot is built on the Zig-bee platform and supports bidirectional, secure data transfer between all the robotic agents. The developed robotic platform has higher probability in tracking and locating objects in lesser time as all the robotic agents work in parallel and share localized intelligence with each other, thereby facilitating better situational awareness. This report outlines the design of the different subsystems for the unmanned ground vehicle.
CHAPTER 1 : BACKGROUND

The scope of the project is to develop an autonomous robotic platform that is capable of performing a diverse range of tasks. The robot may be independently controlled by an operator or may be part of a larger network of swarm robots and may receive top level instructions from a master robot and the designed robot might just function as a slave robot which will carry out designated tasks. Either ways the platform of the robot architecture is expected to remain the same. The complete platform is being developed on a hybrid distributed embedded system comprising of two main controllers: The Raspberry Pi and the PSoC (Programmable system on Chip). The fundamental idea behind using two controllers is to ensure the real-time performance of the entire system.

The robot is built on a differential wheel drive chassis system comprising of four high torque DC geared motors. The two motors on either side of the robot are connected in parallel thereby enabling the control system to control two speeds i.e. of the right and left wheels as a pair. The turning mechanism of the robot is based on the angular velocity arising out of differential velocities of the wheel. The length to breadth ratio of the chassis has been optimized to facilitate turning. The robot is also equipped with suspension system making it capable of navigating through rugged terrain where the obstacles are proportional to the size of the robot.

Two of the motors of the robot are equipped with optical encoders which give feedback on the shaft angle and the number of rotations of the shaft in unit time. The output from the optical encoder is in the form of quadrature pulses which need to be measured by the sensing system. The robot is also equipped with a variety of sensor suites ranging from forward looking IR to avoid obstacles to GPS and magnetometers to give a directional heading.

The role of the PSoC is to measure the data from the encoders and the sensor suite and pass them on to the Raspberry Pi (RPi) for use in the control algorithm. The PSoC communicates with the RPi using the serial interface and handshaking is implemented to ensure the integrity of data. The RPi has all the control algorithms built into it and depending on where the robot needs to navigate it calculates the desired velocity of both the wheels and communicates this information.
to the PWM units which generates the desired PWM on two separate channel which is further given to the motor driver units that finally connects to the motors and implements the speed control of the system.

The block diagram shown in figure 1, elucidates the complete working of the system:

*Fig 1: A block diagram of the prototype robotic platform*
CHAPTER 2 : THE RASPBERRY PI

OBJECTIVE

In this section we investigate how the raspberry pi can be used to develop and integrate ‘scalable software and hardware subsystems’ for an autonomous robotic platform.

SCOPE OF THE RASPBERRY PI IN THE PROJECT

1. To develop a robust interface between the control unit and the digital encoders connected to the motor.
2. To integrate an array of digital and analog sensors with the control unit.
3. To develop a communication stack layer at different nodes and to identify a suitable communication protocol to transmit relevant data between different control unit nodes of the platform.
4. To implement supervisory control of the robotic platform and integrate multiple functionalities like navigation control, terrain mapping based on sensor fusion data.

PROPOSED ARCHITECTURE

The proposed system consists of five functional blocks which include the Raspberry-Pi (RPi), PSoC (Programmable system on chip), Sensor array, GPS and the Motor assembly.

The Raspberry-pi (RPi) which is a pocket-sized single board computer with an ARM11 processor functions as the master controller of the system. Figure 2 shows the different peripherals the board can support. The RPi is responsible for supervisory control of the system. The RPi is interfaced to a slave controller ‘PSoC’ which is a dedicated processor that is interfaced to all the sensor modules and feedback devices in the system. The PSoC captures the sensor fusion data, following which it aggregates the data and attaches a time stamp to it and then wraps the data onto a communication stack and transmits it to RPi. The RPi decodes the COM stack received from the PSoC and takes the necessary control action. The RPi directly manipulates the control signals to the motor drivers and consequently alters the trajectory of the robot. The sensor array
provides necessary environment variables that may assist in the secondary or primary objectives of the mission. The GPS subsystem provides closed loop navigation support to facilitate accurate path trajectory.

**Fig 2:** A complete overview of the Raspberry Pi 2.0 B, the top section shows the pinmap and the lower section shows the location of the different peripherals.
2.1 INTRODUCTION TO RPI

Raspberry Pi as seen in figure 3, is an open source Linux based embedded prototyping platform that has dramatically revolutionized the concept of a single board pocket sized computer. The Raspberry pi is essentially offers all the essential tools and utilities that a desktop computer can offer. The Raspberry pi runs an operating system called the Raspbian which is a modified version of the Linux operating system, this ensures that the price of the platform is low. The Raspberry pi offers significant enhancements over the traditional microcontroller prototyping platforms however, this is at the cost of the real-time behavior of the platform. An application that has a number of tasks and involves the use of an internet enabled platform is the perfect use case of the raspberry pi. The raspberry pi is a versatile platform and supports programming in GUI environments like scratch, python based shell scripting, C, java, PERL and a range of other languages. The raspberry pi connects to the outside world through peripherals like the HDMI which can be integrated to any HDMI enabled display. The raspberry Pi version 2B plus support 4 USB host ports where slave devices such as keyboard, mouse, Wi-Fi-dongle and web camera
can be connected. The limits of the raspberry pi is solely dependent on the imagination of the developers.

2.2 CODING ON THE RPI

2.2.1 WRITING AN OS IMAGE TO RPI

Fig 4: A screenshot of the win 32 disk imager software

1. In order to load an image of the operating system on the SD card you will need to install an imaging utility called Win32DiskImager. It is an open source tool available from the Source forge developers.
2. After inserting the SD card onto the adapter connected to the PC make sure that you know the assigned directory.
3. Run the Win32DiskImager as an administrator and select the raspbian image file then select the corresponding drive and burn the image on the selected drive as shown in figure 4. It is important to note that any mistake in selecting the drive at this stage can have severe impact on the functioning of the PC and can also lead to a system crash.
4. In order to burn the image on the SD card select the Write option and wait till the writing process is complete.
2.2.2 PLUGGING IN YOUR RASPBERRY PI

Fig 5: A closer look at the HDMI output of the raspberry pi

1. Prior to plugging in the RPi ensure that you have all the materials listed above. Thereafter follow the instructions below.
2. Insert the SD card into the designated slot on the RPi.
3. Plug in the HDMI cable as shown in figure 5 to a HDMI enabled display so that you can view the command line interface on the screen and enter the login details as shown in figure 6.

Fig 6: A screenshot of the command line on the Raspberry Pi during start-up

Prepared by Abhishek Roy and submitted to Dr. Lisa Fiorentini (Autumn semester, 2016)
4. The default user name is ‘pi’ and the default password is ‘raspberry’. Note that when you type the password, it will not be displayed on the screen, this is an inbuilt security in the Linux OS.

5. In order to connect the raspberry pi to the internet you must plug in an Ethernet cable connected to an open LAN network.

6. After completing all the above steps you can plug in the power jack on the RPi. This will boot the RPi and you can observe a chain of commands on the display that stops after it requests for the login details.

7. Enter the login details as described above and after the console reaches the second pause and requests for an input command type ‘startx’

8. This will enable the GUI mode and you can see the Raspberry pi GUI environment from which you can carry out all the desired operations.
CHAPTER 3 : MATLAB AND RASPBERRY PI

The aim of the demo is to establish a robust communication channel between the RPi and external devices. One of the possible ways of doing this reliably is by using serial communication as shown in figure 7. In our system we will use RS232 standard for the physical layer.

The system configuration for serial communication is 8 data, no parity, 1 start and 1 stop bit. The communication happens at a BAUD rate of 9600 to minimize BAUD rate error that is prominent in higher frequencies.

The smart sensor is emulated on an 8bit microcontroller that sends data on the serial bus. The raspberry Pi receives the data decodes the data and acknowledges the same after each data packet.

The Serial to USB converter is used to monitor and visualize the data on a computer screen. The additional node does not take any active part in the communication in the bus.
The RPi having operating voltages below that of the emulated smart sensors requires a logic shifter for safe operation. This is typically achieved by using a high impedance buffer which can take 2 supply voltages.

The emulated smart sensor sends serial data on the bus, the raspberry Pi receives and decodes these messages, following which the internal algorithm classifies the appropriate action for the given input. The corresponding units of LEDs are turned on.

**GENERAL INTERFACE DETAILS:**

Raspberry Pi 2 Model B is the platform for the project.

Firmware development of Raspberry Pi is done on Matlab using the matlab support package for raspberry pi.

An Ethernet communication link is used to establish connection with the target (RPi).

The matlab code (.m file) is compiled and run on the computer, which controls the RPi and its peripherals via signals over the Ethernet protocol.

A modified version of the Raspbian wheezy OS recommended by the matlab support package is used to boot the raspberry pi.
CHAPTER 4 : SAMPLE CODES

4.1 GENERAL PURPOSE INPUT OUTPUT

The Raspberry pi features a good number of GPIO pins which can vary depending on the version of the RPi being used. All the GPIO pins are 3.3V compatible and can function as both input and output pins. It is important to note here that since these pins are 3.3V compatible logic level shifters are required when interfacing between the RPi and other 5V compatible devices.

The code below shows how to configure a gpio pin as input and output and how to write digital data on the pin and how to read digital data on the pin.

It’s important to understand that the RPi has different pin numbering layouts which need to be specified in the code. In this report we will use the BCM standard issued by Broadcom.

```python
#!/usr/bin/env python

import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.OUT)

for I in range(0,10):
    GPIO.output(17,1)
    time.sleep(1)
    GPIO.output(17,0)
    time.sleep(1)
```

4.2 SERIAL COMMUNICATION

The RPi features a full duplex serial protocol with BAUD upto 11520. These can be integrated with the RS232 standard or with the RS485 standard and can also be used as TTL devices.
The code below reads a single character from the RX line and print it on the terminal window. After running the code below it is easy to notice that even when there is no data in the RX line the RPi records a junk value.

```python
#!/usr/bin/env python

import time
import serial

ser = serial.Serial(
    Port = '/dev/ttyAMA0',
    Baudrate = 9600,
    Parity = serial.PARITY_NONE,
    Stopbits = serial.STOPBITS_ONE,
    Bytesize = serial.EIGHTBITS,
    Timeout = 1
)

while 1:
    serialdata = ser.read(1)
    print(serialdata)
```

The following code adds an extra line that ensures that the junk values are omitted and the bug in the previous version of the code is fixed.

```python
#!/usr/bin/env python

import time
import serial

ser = serial.Serial(
    Port = '/dev/ttyAMA0',
    Baudrate = 9600,
    Parity = serial.PARITY_NONE,
    Stopbits = serial.STOPBITS_ONE,
)
`Bytesize = serial.EIGHTBITS,`  
`Timeout = 1`  
`)
While 1:`  
`Serialdata = ser.read(1)`  
`If serialdata != ' ':`  
`Print serialdata`
#!/usr/bin/env python

Import time
Import serial

Ser = serial.Serial (
    Port = '/dev/ttyAMA0',
    Baudrate = 9600,
    Parity = serial.PARITY_NONE,
    Stopbits = serial.STOPBITS_ONE,
    Bytesize = serial.EIGHTBITS,
    Timeout = 1
)

Ser.write('H')

While 1:
    Serialdata = ser.read(1)
    If serialdata != 'K':
        While 1:
            Ser.write('A')
            Encoderticks = ser.read(8)
            Print('%s
%(Encoderticks)

4.3 ROBOT LOCOMOTION

The following code implements the open loop navigation of the robot. In this code the raspberry pi uses GPIO channel 20 and 21 for direction control and uses GPIO 18 for PWM control and GPIO23 as input pin.

#!/usr/bin/env python

Import RPi.GPIO as GPIO
Import time
Dc=30

GPIO.setmode(GPIO.BCM)
GPIO.setup(18,GPIO.OUT)
P = GPIO.PWM(18, GPIO.OUT)
P.start(0)
GPIO.setup(20, GPIO.OUT)
GPIO.setup(21, GPIO.OUT)
GPIO.setup(23, GPIO.IN)

While (GPIO.input(23) != 1):
    time.sleep(0.001)

For j in range (1,5)
    # go straight
    GPIO.output(20,1)
    GPIO.output(21,1)
    # ramp up
    For i in range (0,70):
        p.ChangeDutyCycle(i)
        time.sleep(0.05)
        time.sleep(0.2)
    # go left
    p.ChangeDutyCycle(0)
    time.sleep(0.5)
    GPIO.output(20,1)
    GPIO.output(21,0)
    For i in range (30,71):
        p.ChangeDutyCycle(i)
        time.sleep(0.005)
        time.sleep(0.4)

p.stop
CHAPTER 5: ENCODERS DOCUMENTATION

Fig 8: An internal view of the gear mechanism inside the high torque DC motor

The geared motor system obeys a simple equation: \( \text{PCR}_1 \cdot \omega_1 = \text{PCR}_2 \cdot \omega_2 \)

For approximately 74.83 rotations of the base motor shaft the wheel shaft rotates by one revolution, this can be better understood be observing figure 8.

Fig 9: A screenshot of the quadrature encoder pulses from the motor shaft encoders

Courtesy Pololu Robotics and Electronics

Prepared by Abhishek Roy and submitted to Dr. Lisa Fiorentini (Autumn semester, 2016)
1. Channel 1 and channel 2 are outputs of the two Hall Effect sensors attached to the encoder module.

2. Two Hall Effect sensors in the encoder module are redundant, but are for plausibility check, if one fails the system should not be compromised.

3. For one revolution of the base motor, both channel 1 and channel 2 have 12 rising edges and 12 falling edges (per channel) as seen in figure 9.

4. Total edges for one base motor shaft revolution = channel 1 \( \text{RE} \) + channel 2 \( \text{RE} \) + channel 1 \( \text{FE} \) + channel 2 \( \text{FE} \) = 12 + 12 + 12 + 12 = 48.

\[ \text{Fig 10: The product description available commercially} \]

\[ \{(\text{No of rising + falling edges on channel 1}) + (\text{No of rising + falling edges on channel 2})\} \times \text{gear ratio} = \text{Total number of edges for one revolution of wheel shaft} \]

48 \times 74.83 = 3591.84

Note: The underlined section corresponds to one revolution of base motor shaft
Encoder output from one channel is sufficient to compute the instantaneous speed of the robot wheels

12 falling edges = 1 revolution of base motor

1 revolution of wheel = 74.83 base motor shaft revolutions (taken from figure 10)

No. of edges in 1 revolution of wheel = 12 x 74.83 = 897.96

Distance covered by the robot in one revolution of wheel = $2\pi r$ (r = radius of the wheel)

Algorithm to compute speed using a high precision timer and the above data.

Assumption: Wheels have no slip and no sliding i.e. the surface is ideal and friction is uniform.

```
WAIT till falling edge is detected (via interrupt)
If falling edge detected :
    Start Timer : Time = 0
    Check if “falling edge count” == 898
        False:
            increment falling edge count
        True:
            wheel revolutions ++
            falling edge count = 0
    Stop Timer : Time = xx
    break

Speed = $2\pi r$ / Time
```

Total Distance = $(2\pi r \cdot \text{wheel revolutions}) + (2\pi r \cdot (\text{falling edge count} / 898))$

Assumption: Wheels have no slip and no sliding i.e. the surface is ideal and friction is uniform.
CHAPTER 6 : THE PROXIMITY SENSOR

A most commonly used design of a proximity sensor available in market is shown in figure 11.

As you can see, it has a potential divider network, a comparator IC (LM385), IR LED, photodiode and two resistors. Its range is about 10 to 15 cms. It has three pins, one for supply (5 volts), and another for ground, and the last one for output. Based on the preset resistance on POT, the sensor gives a high output for the particular distance proximity.

These sensors are commonly used in the field to robotics, to design line follower, obstacle avoider, wall follower, cliff avoider, etc. robots.

WORKING OF THE PROXIMITY SENSOR

An IR LED is a type of light emitting diode which emits infrared radiations. A photo resistor is a type of photo dependent which has a high output resistance in absence of light. In this circuit LM358 is being used as a comparator between the output of the photo resistor (to the non-inverting terminal) and a reference voltage set by a potentiometer (to the inverting terminal).
The output of the comparator is then connected to a diode as shown in figure 12.
Now a $+V_{CC}$ is obtained when the IRLED and photo resistor face a white platform and zero when they face a black platform. The figure below shows a basic circuit diagram for the following module.

The working is based on the principle of absorption of IR radiations by various objects. The sensing system consists of an IR led and an IR photodiode. The system is arranged such that the IR LED and IR photodiode are placed parallel to each other. Whenever an object comes in front of the system, the object absorbs a certain portion and reflects the remaining IR radiation. The emanated IR radiation is received on the photodiode which is kept in a voltage divider.
arrangement. The output of the photodiode is fed to the comparator which gives a bi-state output. The distance till which the sensor can detect an obstacle can be increased by varying the resistance of the variable resistor connected to the non-inverting terminal of the op-amp.

*Fig 12: A schematic of the IR proximity detection sensor*
CHAPTER 7 : THE PWM GENERATION SYSTEM

In order to control the speed and the direction of the robot it is necessary to control the RPM of each wheel of the robot separately. Hence, it is necessary to generate two separate PWM signals using Raspberry pi (for the two MOSFET drivers). The system consists of a Raspberry pi connected to a MOSFET motor driver IRF8734PbF. The PWM outputs from the Raspberry pi are applied as control signals to the motor driver based on the feedback received from the speed encoder.

There are two ways of generating a PWM signal in Raspberry Pi

1) Hardware PWM:
   - A dedicated GPIO is used for generating the PWM signal
   - In RPi, pin 1 is the only hardware pin available
   - Provides a very good resolution (1 microsecond)
   - Low jitter

2) Software PWM:
   - All GPIO pins can be used for generation of PWM signal
   - Resolution is less (100 microsecond) when compared to Hardware PWM
   - High jitter

There are several physical constraints for the system for instance, the motor has two MOSFET drivers and this requires the generation of 2 separate PWM signals from Raspberry pi. Since Raspberry pi has only a single hardware PWM, we might want to use the software PWM pins available in RPi for generating the other PWM output. I2C and SPI enabled smart chips like the Adafruit 16 channel PWM driver can be used and enhances the PWM capability of raspberry pi.

The PWM frequency limitations are discussed in this section. The Maximum resolution (Tres) that can be provided by RPi for a software PWM is 100 us and if we require very fine changes in the PWM signal such that a 1% change in duty cycle change is required, the default PWM range (R) is 100
\[
\frac{1}{R} \times \frac{1}{f} = F = \frac{1}{R \times T_{res}} \\
= \frac{1}{100 \times 100 \times (10)^{-6}} = 100 \text{ Hz}
\]

This gives a default PWM frequency of 100 Hz. As the minimum pulse width can’t go below 100 us and given the range constraint, the frequency is limited to the order of Hertz. For a hardware PWM, maximum resolution (Tres) is 1 us. The default PWM frequency is 10 kHz and the range is 100. To get a higher frequency, we can lower the range or vice versa. However, for precise closed loop operation like motor speed control, a high range is desired. The maximum switching frequency of the MOSFET driver is 40 kHz and the minimum switching frequency is 10 kHz. The operating frequency of the MOSFET is within the range of 10 kHz to 40 kHz.

The PWM signal should be much greater than the resonant frequency of the motor which is usually around 50 Hertz i.e. \( F > \text{fres} \) of the motor. This is because if the frequency of PWM signal is in the order of Hz, whenever there is a change in duty cycle such that the frequency applied to the driver circuit is close to the resonant frequency, it will lead to interference and cause damage to RPi and might also stall the motor. We will use 10 kHz, which is the default PWM frequency of hardware PWM. Therefore to conclude, hardware PWM is preferred over software PWM owing to its better resolution and better operating frequency range.

**HARDWARE PWM GENERATION USING I2C INTERFACE**

The Raspberry Pi has only one hardware PWM pin and we require two independent PWM pins - one for each motor driver. To get a resolution and frequency range same as that of a hardware PWM pin and which meets our requirements, we go for a PWM driver that uses I2C for communication.

**I2C Definition:**

I2C, known as Inter-Integrated Circuit is a series bus that connects multiple devices together and enables them to communicate with each other. The devices are termed as masters and slaves based on the operations they perform. Usually, there is one master and multiple slave devices
are connected to the master. The communication uses only two signal wires (Serial Data and Serial Clock) to exchange information.

**Configuring and installing I2C in Raspberry Pi:**

Raspberry pi is provided with two I2C interfaces (two SDA and SCK pins). We have made use of the pins 3 and 5 for SDA and SCK respectively. Multiple devices can be connected to the Raspberry Pi using the I2C interface. Each device can be accessed by a unique address associated with them.

**Installing the i2c tools utility**

The following commands are entered to enable the System Management Bus (SMBus) and install the I2C tools. The SMBus is a subset of I2C protocol and provides commands for the I2C.

```bash
sudo apt-get install python-smbus
sudo apt-get install i2c-tools
```

**Installing the Kernel Support**

The command `sudo raspi-config` is run to install the i2c support for ARM core and Linux Kernel. Once the raspberry config window is open, select

```bash
advanced options → A7 i2c and enable the ARM interface.
```

**Testing the I2C:**

Following command should be entered in the command line window if Raspberry Pi model B is used.

```bash
sudo i2cdetect -y 1
```
Figure 13 tells us that two I2C addresses 0x40 and 0x70 are in use. The user can use any of them.

**Adafruit 16-channel, 12-bit PWM Driver:**

A 16-channel, 12 bit PWM driver is used to generate the PWM signals for both the motors. The driver is capable of generating PWM signals up to 1 MHz. As discussed before, due to the lack of availability of more than one hardware PWM channel in the Raspberry pi, the multi-channel PWM driver is required to produce necessary PWM signals.

5) **Connecting Raspberry Pi to the PWM driver:**

**Wiring:**

PCA9685 is the chip that is used as the driver which is powered with the voltage of 3.3V from the Raspberry Pi. Since we are giving the PWM signal to a motor driver and the operating voltage of the motor driver is also 3.3V, the voltage that is given to power the motor driver, that is, V+ is also given to be 3.3V.

Figure 14 shows the connections from the Raspberry Pi to the PWM Driver.
Fig 14: A schematic of the PWM generation system

HARDWARE PWM CODE:

```python
#!/usr/bin/env python
from PWM_Driver import PWM
import time
pwm = PWM(0X40)
pwmMin = 50
pwmMax = 1000

def setpwmPulse(channel, pulse):
pulseLength = 1000000
pulseLength /= 60
pulseLength /= 4096
pulse *= 1000
pulse /= pulseLength
pwm.setPWM(channel, 0, pulse) // The pulse width of channel 0 is set to pwmMin
pwm.setPWMFreq(60)                    // The frequency is set to 60 Hz
while (True):
pwm.setPWM(0,0,pwmMin)           // The pulse width of channel 0 is set to pwmMin
time.sleep(5)
pwm.setPWM(0,0,pwmMax)           // The pulse width of channel 0 is set to pwmMax
time.sleep(5)
pwm.setPWM(1,0,200)                // The pulse width of channel 1 is set to 200
time.sleep(5)
pwm.setPWM(1,0,3000)                // The pulse width of channel 1 is set to 3000
time.sleep(5)
```

Prepared by Abhishek Roy and submitted to Dr. Lisa Fiorentini (Autumn semester, 2016)
The Adafruit PWM Driver has a default library which is being imported here to make use of the PWM operations. The link for the same is attached below:

https://github.com/adafruit/Adafruit-PWM-Servo-Driver
Library/blob/master/Adafruit_PWMServoDriver.cpp

The function `pwm.setPWMFreq(frequency value)` is used to set the appropriate PWM frequency required.

The function `pwm.setPWM(channel, on, off)` is used to set the pulse width to appropriate value at the specified channel. Since the PWM driver has a 12-bit resolution, the on and off value can be set to a range of values from 0 to 4095.
CHAPTER 8 : SYSTEM SIMULATION

The system dynamics of the robot is simulated in Matlab to develop a linear control system to control the trajectory of the robot based on sensor feedback from the encoders. The system simulation comprises of four major subsystems which include: (1) The Plant, (2) The Measurement block, (3) The Waypoint generation unit and (4) The Control block. Figure 15 shows the different subsystems and their interconnections.

The first block or the plant subsystem models the robot dynamics using a point mass model. This block receives two parameters (VL and VR) as input from the control block. The two parameters determine the velocity of the left and the right wheel, which is physically (at the hardware level) controlled by the speed controller of the robot.

The second block which is the measurement block imitates the behavior of the system in real-time. The Plant generates the state parameters of the robot which include position (X, Y) and orientation (theta). However, in reality these values generated by the point mass model cannot be accurate and is never free from all errors, hence in order to simulate this real time behavior the measurement block adds white Gaussian noise to each of the channels.

The third block which is the Waypoint generation block determines the target coordinates that the robot must eventually navigate to. This must be given from a high level operator. The high level operator in our case is the user and hence, the waypoints is hardcoded into the system. An important functionality of this block is to determine whether or not the robot has reached within an acceptable distance of the target waypoint and update the next waypoint.

The final block is the control block determines how much velocity should be imparted to both the wheels in order to steer and navigate the robot to the target waypoint.
Fig 15: A Simulink block diagram of the robot dynamic model, path planning and trajectory control system for the robot

In the next section, we will look into each of these blocks individually and analyze all their details and how each of them are interconnected.
8.1 THE PLANT

The simulation uses the point mass model to simulate the motion of the robot. This is acceptable since the robot is not very large in dimension and the length to breadth ratio is close to unity, hence within margin of acceptable error this assumption will not have a large impact on the position estimation accuracy. Figure 16 shows the plant subsystem and its implementation.

![Diagram](image)

**Fig 16: The implementation of the robot dynamics as the plant subsystem**

The following system of equations form the point mass model:

\[
\frac{\partial x}{\partial t} = \frac{V_R + V_L}{2} \sin \theta(t)
\]

\[
\frac{\partial y}{\partial t} = \frac{V_R + V_L}{2} \cos \theta(t)
\]

\[
\frac{\partial \theta}{\partial t} = \frac{V_L - V_R}{b}
\]

It is important to note that all the derivatives listed above are a function of time and hence the system must keep updating them. It is important to understand that the update rate of the orientation must be higher than that of the position.

The position of the robot is a function of the velocities of the wheels \((V_L\) and \(V_R))\) and the orientation \((\theta)\). Hence, if there is an attempt to calculate the new position without calculating the orientation prior to it then the estimation results will not be accurate as they will use the orientation from the previous instant of time.
The orientation angle $\Theta$ is obtained by integrating the rate of change of orientation angle or the angular velocity, $\dot{\Theta}$. We know that integration of an estimated parameter only accumulates any existing noise in the estimation, hence the orientation angle that is obtained from this process will have a lot of noise associated with it.

Since, the estimation of the position of the robot also indirectly depends on the orientation angle a lot of error is introduced by this too, this will be evident during the hardware trials of the robot.

The derivative if the $x$ and $y$ position of the robot can be estimated from: $V_L$ and $V_R$. However, we need to estimate the position and not the derivative of the position hence there are two integrators in the plant model that calculate the position of the robot.

The estimation of the position of the robot has two sources of accumulating error. One from the error accumulation during calculation of the orientation angle and the second when the derivative of the position needs to be integrated.

The coordinate transform block converts the compass coordinate frame to the Cartesian coordinate frame. The compass coordinate frame of reference is used in a number of digital devices like the GPS, however a lot of the trigonometric calculations become cumbersome in this frame of reference and proper care should be taken to ensure when a particular trigonometric function becomes positive or negative. Hence, for simplicity we convert the GPS coordinate frame to the compass coordinate frame. The equations derived above are for the compass coordinate frame and therefore the conversion.

Hence, the output of the plant tells us how the state variable of the robot i.e. $X$, $Y$ and $\Theta$ will change based on the velocities of the two wheels $V_L$ and $V_R$. As discussed above although there exists a lot of sources of error, the final control system does well to converge the estimation to the desired result. In real-time hardware trials the user can expect acceptable deviations which can be eliminated with some form of position feedback.
8.2 THE MEASUREMENT BLOCK

The measurement block as shown in figure 17 adds random noise to the state variables generated by the plant. This can be physically accounted for factors like non-linearity in the system and approximation of the robot as a point mass model.

Fig 17: The addition of noise to each measurement makes the plant response more realistic

The measurement block accepts three additional parameters from the users which include the enable and the gains for the position estimation error and the angle estimation error. In some systems the yaw rate sensor might generate the angle measurement by integrating the angular velocity, and the encoders on the motor might generate the position estimation, mostly the yaw rate sensors are not as accurate as the encoders due to lower resolution and large measurement errors due the effect of temperature and variable loading configurations, In such systems the user has the option of simulating this behaviors by increasing the gain for the angle measurement alone without affecting the position estimation error. In order to only test the plant behavior without the additional error the enable can be set to zero.
8.3 THE WAYPOINT GENERATION UNIT

This subsystem tracks the current coordinates of the robot (x_meas and y_meas) and accordingly determines whether or not the robot is reasonably close to the target and if it is close to the target then it updates a new set of coordinates as target coordinates which the robot must navigate to. This subsystem is implemented as a Matlab function in Simulink. This block also takes in the array of target coordinates (X_Waypoints and Y_Waypoints) and the number of waypoints (Size_Waypoints) in order to determine the next coordinate that the robot must navigate to. In order to determine the acceptable zone within which the robot is considered to have reached the current destination an additional parameter (Error_zone_radius) is taken as input, this value determines the radius of the circle within which the robot is expected to reach. If the current coordinates of the robot lies inside this circle then the waypoints for the robot’s next target coordinates are updated. It’s important to note that if this value is kept very small then the robot will take a longer time to reach the point and sometimes if the point is such that the point is not reachable then the robot might exhibit oscillatory behavior. Hence, it is imperative that the robot error zone radius be proportional to the size of the robot itself. The output of the block is the target coordinates which are inputs to the control block and behave like set points in the control systems framework. The block also tells the controller if the final destination is reached so that the controller realizes that no further control action is required as the final destination is reached.

```matlab
function [x_dest,y_dest,reached] = fcn(x,y,Path_x_req,Path_y_req,err_zone_rad,N)
    reached = 1;
    persistent target_flag;
    if(isempty(target_flag))
        target_flag = zeros(1,10);
        target_flag(1) = 1;
    end

    %find the index of highest 1 in current target matrix
    i=1;

    while((i<N+1)&&(target_flag(i)==1))
        i=i+1;
    end

    i=i-1; % This is that index
```
dist_target = sqrt((Path_x_req(i)-x)^2 + (Path_y_req(i)-y)^2);

if(dist_target < err_zone_rad)
    if(i<N)
        i=i+1;
        target_flag(i)=1;
    end
    if(i==N)
        reached=0;
        target_flag(N)=1;
    end
end

x_dest = Path_x_req(i);
y_dest = Path_y_req(i);

One important implementation detail here is the use of persistent variables which are analogous to the static variable is C. In order to maintain an awareness of all the waypoints that the robot has already covered the history of the details should not be erased on each cycle of Simulink run hence, the target flag is declared as a persistent variable. So each flag position corresponds to a particular coordinate when the coordinate is reached the flag index is rewritten with 1 and the next waypoint is located by finding the lowest index of 0.

**CATMULLROM SPLINES**

Given a set of waypoints the shortest path between the adjacent waypoint would be a straight line joining them, however if the robot trajectory were to be such a zigzag fashion it would lead to a highly oscillatory motion and give rise to lot of vibrations. In order to avoid this usually waypoint generation involves fitting a spline as shown in figure 18. There are different types of spline and for our application the Catmullrom spline is the best choice. Some interesting features of this particular class of spline is that all points on the generated spline have their first derivatives as a continuous function and the spline will always pass through the initial target waypoints. The only difference that the spline will make is that it determines a curved path whose curvature can be controlled and this makes the robot motion smooth and an optimum curvature can be identified based on the set of points. Its important to note that in order to generate a spline the function must have at least four input points hence a look ahead of two waypoints is essential. Typically of the four waypoints the first waypoint corresponds to a position in the past and the
second waypoint corresponds to the current position and the last two way points correspond to looking ahead in future. The curvature of the trajectory can be controlled by modifying the constant parameter in the code attached below:

```matlab
clc; clear all;
Path_x_req = [-1; 0; 3; 4; 5];
Path_y_req = [-1; 0; 25; -25; 16];
plot(Path_x_req,Path_y_req,'--rx','MarkerSize',15);

Path_x = [-1; 0; 3; 4];
Path_y = [-1; 0; 25; -25];
hold on;
k=0.5;
Q = [0 1 0 0; -k 0 k 0; (2*k) (k-3) (3-(2*k)) -k; -k (2-k) (k-2) k];
t = transpose(0:0.1:1);
P = ones(size(t,1),4);
P(:,2) = P(:,2).*t;
P(:,3) = P(:,3).*(t.*t);
P(:,4) = P(:,4).*(t.*t.*t);
Trajectory_x=(P*Q)*Path_x;
Trajectory_y=(P*Q)*Path_y;
plot(Trajectory_x,Trajectory_y,'k.','MarkerSize',4.6)
hold on;
Path_x = [0; 3; 4; 5];
Path_y = [0; 25; -25; 16];
Trajectory_x=(P*Q)*Path_x;
Trajectory_y=(P*Q)*Path_y;
plot(Trajectory_x,Trajectory_y,'k.','MarkerSize',4.6)
```

**Fig 18:** The smoothening effect of the trajectory with catmullrom splines
8.4 THE CONTROL BLOCK

This block determines the trajectory control by estimating the desired velocities in each of the wheels in order to reach the target waypoint. This block receives inputs from the measurement block which are the measured state parameters of the robot i.e the position and orientation of the robot, it also receives parameters from the waypoint generation block which include the target coordinates and a flag that determines whether the final checkpoint is reached. This block also receives two more inputs which include the constant forward velocity (Const_vel) and the proportional gain of the controller (Kp). The output from this block is the calculated desired speeds of the two wheels.

CONTROL ALGORITHM

Let us assume that the robot is at $P_1 (X_1, Y_1, \Theta_1)$ and wants to go to $P_2 (X_2, Y_2, \Theta_2)$ as shown in figure 19. For our analysis we will use the compass coordinate system.

*Fig 19: The current position is $P_1$ and the next position is $P_2$. $V_1$ and $V_2$ are the position vectors.*
The objective of the control algorithm is to minimize the error angle ($\theta_e$)

$(\vec{V}_1)$ is a unit vector in the current direction, hence $\vec{V}_1$ can be written as

$$\vec{V}_1 = \sin(\theta_1) \hat{i} + \cos(\theta_1) \hat{j}$$

$$\|\vec{V}_1\| = 1 \quad \text{... since, } \vec{V}_1 \text{ is a unit vector}$$

$(\vec{V}_2)$ is the vector connecting current position and destination

$$\vec{V}_2 = P_2 - P_1$$

$$\vec{V}_2 = (X_2 - X_1)\hat{i} + (Y_2 - Y_1)\hat{j}$$

Let us find a unit vector along $(\vec{V}_2)$

$$\vec{V}_2 = \frac{\vec{V}_2}{\|\vec{V}_2\|} = 1$$

$$\vec{V}_2 = \frac{(X_2 - X_1)\hat{i} + (Y_2 - Y_1)\hat{j}}{\sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}}$$

The dot product of $(\vec{V}_1)$ and $(\vec{V}_2)$ is defined as

$$(\vec{V}_1, \vec{V}_2) = \|\vec{V}_1\|\|\vec{V}_2\| \cos \theta_e$$

Since, both are unit vectors

$$(\vec{V}_1, \vec{V}_2) = \cos \theta_e$$

$$\theta_e = \cos^{-1}(\vec{V}_1, \vec{V}_2)$$

$$\theta_e = \frac{\sin(\theta_1) \times (X_2 - X_1)}{\sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}} + \frac{\cos(\theta_1) \times (Y_2 - Y_1)}{\sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}}$$

However, it must noted that when $P_1 (X_1, Y_1, \theta_1)$ and $P_2 (X_2, Y_2, \theta_2)$ are interchanged the error angle ($\theta_e$) still remains the same. To account for this we need to impart a direction sense to the error angle ($\theta_e$) this is done using the cross product $(\vec{V}_1 \times \vec{V}_2)$
If \((\vec{V_1} \times \vec{V_2})\) is positive the direction of motion is anticlockwise or in other words the right wheel must have higher angular velocity than the left wheel.

If \((\vec{V_1} \times \vec{V_2})\) is negative then the direction of motion is clockwise and the left wheel should have higher angular velocity than the right wheel.

\[
(\vec{V_1} \times \vec{V_2}) = \frac{\sin(\Theta_1) \times (Y_2 - Y_1)}{\sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}} - \frac{\cos(\Theta_1) \times (X_2 - X_1)}{\sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}}
\]

Depending on the result of the cross product \((\vec{V_1} \times \vec{V_2})\) the error angle \((\Theta_e)\) will either have a clockwise or an anticlockwise sense.

If, \((\vec{V_1} \times \vec{V_2}) > 0\) then the error angle \((\Theta_e)\) is negative or in clockwise sense.

In this algorithm, both \(V_L\) and \(V_R\) are assumed to have a constant offset velocity.

\[
V_L = V_{\text{offset}} + K_P \cdot \Theta_e
\]

\[
V_R = V_{\text{offset}} - K_P \cdot \Theta_e
\]

**SYSTEM IMPLEMENTATION**

It is important to note that the control system implemented in this simulation is a steering control system. How this works is that the bot has a constant forward velocity and if no change in direction is required it continues to move in the initial direction however, if it is determined that the optimal way to reach a particular waypoint is by altering the current direction then the steering control system takes control and modifies the velocity on both the wheels by an equal factor and this results in the robot taking a turn due to differential wheel speeds.

The control block is also implemented as a Matlab function:

```matlab
function [VL,VR] = fcn(x,y,theta,const_vel,Kp,x_dest,y_dest,reached)
%#codegen

%Unit vector along the direction of current heading in compass coordinate
%system is = Sin(theta) i + Cos(theta) j
%Vector along the required direction is = (x_dest-x) i + (y_dest-y) j

V2_mag = sqrt(((x_dest-x)^2)+((y_dest-y)^2));
```
V1_dot_V2_x = \sin(\theta) \times \frac{(x_{\text{dest}} - x)}{V2\_mag}; \quad \%\text{both are unit vectors}
V1_dot_V2_y = \cos(\theta) \times \frac{(y_{\text{dest}} - y)}{V2\_mag}; \quad \%\text{both are unit vectors}

\text{error\_angle} = \arccos \left( V1\_dot\_V2\_x + V1\_dot\_V2\_y \right); \quad \%\text{cos inverse dot product}

\text{angle\_dir} = (\sin(\theta) \times \frac{(y_{\text{dest}} - y)}{V2\_mag}) - (\cos(\theta) \times \frac{(x_{\text{dest}} - x)}{V2\_mag})

\text{if} (\text{angle\_dir}>0)
    \text{error\_angle} = \text{error\_angle} \times (-1);
\text{end}

V_L = (\text{const\_vel} + (K_p \times \text{error\_angle})) \times \text{reached};
V_R = (\text{const\_vel} - (K_p \times \text{error\_angle})) \times \text{reached};
8.5 STEPS TO BEGIN SIMULATION

In order to start the simulation you will need to run the init.m file. The init file invokes the Simulink model and passes the Simulink model the required parameters.

The user must initialize the waypoints in the arrays X1 and Y1 and enter value for V, Kp and R_Err.

The robot will start from the origin and navigate through each of these waypoints.

In order to change the initial coordinates of the robot the integrators for X,Y and Θ can be initialized to the desired non zero values.

The init invokes the Simulink model and the Simulink model runs the simulation across the control blocks for the specified time and then return the measured trajectory parameters to the workspace from which the trajectory is plotted.

It is important to note that the working directory of the project must the location of all the desired files. The Simulink model must be in the same folder as that of the init.m file and the working directory should be set as this particular file. The simulation time should also not be too small or the robot may not reach the destination.

close all;
clear all;
clc;
X1 = [0, 0, 20, 33, 50, 58, 50, 30, 20, 20];  % x waypoint
Y1 = [20, 40, 43, 46, 57, 78, 95, 110, 120, 140];  % y waypoint
V = 0.1;  % fixed bias velocity of the robot
Kp = 1;  % proportional gain for error angle
R_err = 2;  % radius of position error circle
size_N = size(X1,2);
sim('closedloop_system_real')

hold on;
grid on;
comet(X,Y)
hold on;
CHAPTER 9 : VALIDATION OF THE ENCODERS AND POSITION ESTIMATION ALGORITHM

It is imperative that no two manufactured components can have the exact same physical properties and motors for robotics hobbyist don’t go through a very stringent quality check and hence a number of parameters for two motors with the same ratings can vary. It is the role of the control algorithm to account for all these non-linearity in the control algorithm.

In order to validate the encoders the robot was placed on a bench top such that wheels were freely suspended and the raspberry pi that controls the motors were programmed with a fixed duty cycle for a certain period of time such that a particular point on the wheel of the robot completed exactly integer multiples of rotation. The PSoC which polls the data from the encoder measured the number of pulses generated by the encoder and returned the sum of all ticks. The average of the sum of ticks over the total number of rotations gives the number of ticks per rotation. This is a crucial factor that determines the error in the estimation of the angular orientation of the robot.

In order to account for the loading on wheels and any additional non-linearity that that might add the raspberry pi was preprogrammed with a number of profiles and the extent of deviation from the expected and observed value was measured. The data acquired was and measured was transferred to MATLAB for further analysis. The following section discusses the codes and the results.

%Data1 (50,50,6) + (50,30,3)
%Y-axis 300 ; X-axis 40.5 | 23 ; Theta 60 degrees (approx)

%Data2 (50,50,6) + (30,50,3)
%Y-axis 300 ; X-axis 24.5 | 30 ; Theta 60 degrees (approx)

%Data3 (50,50,6) + (30,60,4)
%Y-axis 285 ; X-axis 100 ; Theta -90+ degrees (approx)

%Data4 (50,50,6) + (60,30,4)
%Y-axis 275 ; X-axis 109 | 83 ; Theta -90+ degrees (approx)

%Data5 (50,50,8)
%Y-axis 303.5 | 300 ; X-axis Negligible 10 ; Theta 0 degrees (approx)
%Data6 (62,38,10)
%Y-axis -63 | 50 ; X-axis 153 | 180 ; Theta 225 degrees (approx)

%Data7 (50,50,2) (40,60,3) (60,30,7) (40,40,2) (30,60,2)
%Y-axis 174 ; X-axis 137 ; Theta 90 left degrees (approx)

clc;
clear all;

%constants
RADIUS = 6;
WIDTH = 26;
%TLMAX  = 3506;
%TRMAX  = 3580;
TLMAX = 3600;
TRMAX = 3600;
DT = 0.5;

%initial conditions
X(1) = 0;
Y(1) = 0;
Phi(1) = 0;

%import data from psoc
[ticks_right,ticks_left] = textread('Data3A.txt','%u %u');
%[ticks_right,ticks_left,a,b,c,d,e,f] = textread('ticksdata.txt');

ticks_left  = 5000 - ticks_left
ticks_right = 5000 - ticks_right

for i = 2:(size(ticks_left,1))
    SL(i) = ((2*pi*RADIUS*ticks_left(i))/ TLMAX);
    SR(i) = ((2*pi*RADIUS*ticks_right(i))/ TRMAX);
    if (SL(i) ~= SR(i))
        K = (SL(i)+SR(i))/(SL(i)-SR(i));
        del_Phi(i) = ((SL(i) - SR(i))/ WIDTH);
        Phi(i) = del_Phi(i) + Phi(i-1);
        X(i) = X(i-1) - ((WIDTH/2)*K*(cos(Phi(i))-cos(Phi(i-1))));
        Y(i) = Y(i-1) + ((WIDTH/2)*K*(sin(Phi(i))-sin(Phi(i-1))));
    end
    if (SL(i) == SR(i))
        Phi(i) = Phi(i-1);
        X(i) = X(i-1) + (((SL(i) + SR(i))/2*DT)*sin(Phi(i)));
        Y(i) = Y(i-1) + (((SL(i) + SR(i))/2*DT)*cos(Phi(i)));
    end
    Deg(i) = (Phi(i)*180)/pi
end

t=1:1:size(ticks_left,1);
figure(1);
plot(X,Y,'-*','LineWidth',2,'MarkerSize',15,'MarkerEdgeColor','k','MarkerFaceColor',[0.5,0.5,0.5])
title('path trajectory for (x,y)')
xlabel('x distance')
ylabel('y distance')
The initial comments describe the duty cycle on each of the wheels and the duration and the observations of the position and orientation. These are compared to those estimated using the position estimation algorithm as shown in figure 20.

**Fig 20: The plots of the data collected during the validation of the position estimation algorithm**
CHAPTER 10 : RESULTS AND CONCLUSION

The robot prototype was developed and their subsystems i.e. The PSoC, the peripherals and the RPi were independently designed tested and then integrated. The robot system dynamics was simulated on MATLAB and a control strategy for trajectory control based on encoder feedback was designed, tested and validated.

The simulation results for two sets of waypoints are shown below in figure 21 and figure 22:

Fig 21: The simulation results for a set of waypoints. The large red crosses represent the waypoints and the blue line is the trajectory of the robot starting from the origin.
Fig 22: The simulation results for a different set of waypoints. The large red crosses represent the waypoints and the blue line is the trajectory of the robot starting from the origin.

Clearly the robot can track all the waypoints and the trajectory control system is capable of making both concave and convex turns and hence, the control algorithm designed is robust enough for most of the real-time scenarios the robot might encounter.

Future scope of this project can take into account the geometry and dimension of the robot and develop a system dynamic model that is not based on the point mass model. The curvature of the catmullrom spline can be optimized for each of the paths with some cost function depending on path curvature being maximized over that path.

Adding an image based feedback into the system can help correct the accumulated error due to noisy measurements. Adding a GPS can also help solve the problem of the accumulating error,
Hence, in a realistic scenario there is a need for sensor fusion based position estimation algorithms that can more accurately estimate whether the waypoint have been reached. The current algorithm also doesn’t take into account obstacle avoidance which can be integrated.

![Image of the final constructed robotic prototype](image)

**Fig 23: A top angular view of the final constructed robotic prototype**

Figure 23 shows the final robot prototype robot that featured a two controller distributed embedded system platform comprising of the Raspberry Pi and the PSoC.

The robot was capable of position estimation, path planning, trajectory control and obstacle avoidance. Future work on the robot will focus on integrating the robot with other similar robots and aerial vehicles into a swarm network.
REFERENCES

1. “Localization and control of unmanned ground vehicles using quadrature encoders and pure pursuit algorithms”, Abhishek Sharma, June 2015

2. “Self-Navigation Control Based on Unmanned Vehicle in Non-obstacle field”, Zhuo Xu

3. “PWM generation and control in Raspberry Pi”, Anand Vignesh, Bhargavi Govindrajan
